МГАПИ

МОСКОВСКАЯ ГОСУДАРСТВЕННАЯ АКАДЕМИЯ
ПРИБОРОСТРОЕНИЯ И ИНФОРМАТИКИ

Кафедра ‘Персональные компьютеры и сети’

В.М.Баканов

Разработка прикладных программ для ОС WINDOWS с помощью интегрированных сред Delphi / C++Builder

Москва
2000

АННОТАЦИЯ

Данное пособие предназначено для студентов II÷V курсов, обучающихся или выполняющих лабораторные и практические задания по курсу 'Прикладное программирование' или создающих программное обеспечение (ПО) общего класса в среде операционный системы (ОС) WINDOWS (все версии).

Изложены основы создания WINDOWS-интерфейса прикладных программ с помощью RAD-систем (Rapid Application Design) **Delphi** и **C++Builder** фирмы Borland. Рассмотрены вопросы практического применения интерфейсных элементов WINDOWS для ввода и вывода пользовательских данных и управления выполнением программы. Приведена информация о **Delphi**-подобной RAD-системе **Kylix** (http://www.borland.ru/kylix/index.html) для работы в среде ОС Linux.

Данное пособие НЕ ЯВЛЯЕТСЯ полным и всеобъемлющим руководством по созданию прикладных программ с помощью **Delphi** и **C++Builder** (для изучения конкретных вопросов следует обратиться к соответствующим литературным источникам); однако после изучения пособия имеющий некоторый опыт создания ДОС-программ, программирования на языке **Pascal** и **C++** и навигации в WINDOWS пользователь вполне сможет разрабатывать простые интерфейсы в стиле WINDOWS.

Требуемое аппаратное обеспечение для проведения работ - ПЭВМ класса IBM PC не хуже AT/586 с размером ОП более 32 Мбайт с твердым диском объемом не менее нескольких Гбайт, дисплей класса не хуже VGA. Пакеты **Delphi** и **C++Builder** требуют более 150÷250 Мбайт дискового пространства. Последняя версия данного методического пособия может быть получена в виде файла http://pilger.mgapi.edu/metods/borland.zip.

Автор: доцент, к.т.н. Баканов В.М.
Рецензент: доцент, к.т.н. Гонихин О.Д.
Научный редактор: профессор, д.т.н. Михайлов Б.М.
Работа рассмотрена и одобрена на заседании кафедры
ИТ-4 МГАПИ _____ 2001 года.

Заведующий кафедрой ИТ-4 профессор, д.т.н. Б.М.Михайлов

Предыдущие издания:	©	Разработка WINDOWS - интерфейса прикладных программ в среде Borland DELPHI. -М.: МГАПИ, 1997, -63 С.
	©	Разработка WINDOWS - интерфейса прикладных программ с помощью интегрированных сред DELPHI / C++Builder. -М.: МГАПИ, 1998, -69 С.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Известно, что до 50÷70% времени на создание (с помощью традиционных средств) программного обеспечения (и часто столько же по объему) приходится на разработку интерфейсной части программы (функционирование окон ввода и вывода данных, управление режимами функционирования программы etc). Современные операционные системы (например, ОС WINDOWS, [1]), снабженные графическим интерфейсом и набором стандартных интерфейсных элементов, значительно облегчают нелегкий (и часто неприятный) процесс разработки интерфейса пользователя.

С другой стороны пользователь не отказался бы от механизма, облегчающего рутинную работу по программированию повторяющихся участков кода - таймеров, блоков обращения к базам данных, системам статистической обработки данных и разрешения стандартных математических уравнений, подсистемам сетевого д**о**ступа и многих других.

Начало 90-х годов ознаменовалось значительным ростом аппаратных возможностей персональных ЭВМ (ПЭВМ) и, в связи с этим, массовым переходом к использованию оболочки WINDOWS разработки Microsoft Corporation, снабженной дружественным графическим интерфейсом и обладающей возможностью многозадачности. Полное соответствие международным стандартам **CUA** (*Common User Access*), огромный набор системных процедур WINDOWS и отсутствие ограничений на создание собственных (концепция DLL-библиотек), возможность д**о**ступа к оперативной памяти практически неограниченного объема и многие другие поддерживаемые возможности обусловили применение WINDOWS почти на каждом оснащенном ПЭВМ рабочем месте. Дальнейшее перерастание оболочки WINDOWS'3.1 в полноценную операционную систему WINDOWS'9x/WINDOWS’NT еще более увеличило интерес к ней (не всегда, впрочем, полностью оправданный).

Однако переход к новому поколению программных средств разработки прикладных пользовательских программ значительно (до 3÷5 лет) задержался - программирование под WINDOWS на 'старом добром' **C** (или **Pascal**'е) более чем нерационально, работа на **C++** непроста даже с использованием библиотек классов **MFCL** (*Microsoft Foundation Class Library*) фирмы Microsoft Corp. и **OWL** (*Object Windows Library*) фирмы Borland. Явно необходим был новый подход к созданию программного продукта уровня разработчика, что позволяло самому широкому классу программистов включиться в бесконечную гонку создания собственных WINDOWS-приложений.

Новый подход действительно был разработан; соответствующие системы были названы **RAD** (*Rapid Application Design* - ср**е**ды быстрой разработки приложений); основа этих систем - соответствующая библиотека классов **VCL** (*Visual Components Library* - библиотека визуализируемых классов). Конечно, **RAD**-системы не появились на пустом месте - одним из их 'прародителей' была широко известная библиотека **Turbo Vision** (имеющаяся в **C**- и

Pascal-вариантах разработки Borland), полностью основанная на плодотворнейших идеях программирования конца 20 века - объектно-ориентированном программировании (ООП). После внедрения **Turbo Vision** оставался лишь один прямо напрашивающийся шаг - добавление возможностей работы с объектами как с визуальными сущностями именно на этапе проектирования интерфейса - **DesignTime** (используя естественную для WINDOWS концепцию 'перетаскивания' - *Drag&Drop*), и этот шаг был сделан в основанных на **VCL**-подходе **RAD**-системах.

В первом пятилетии 90-х годов приблизительно одновременно появились первые RAD-продукты - **Visual Basic** и **Visual C++** (Microsoft Corp.), **Delphi** (Borland), **Optima C++** и некоторые другие.

С точки зрения автора (давнего приверженца интегрированных сред фирмы Borland), наиболее удобным **RAD**-пакетом являются системы класса **Delphi** / **C++Builder** (http://www.borland.com.ru), обладающая удачным соотношением между сложностью (а значит, и определенной негибкостью при использовании) и широтой набора объектов (как визуальных, так и невизуальных, что как раз и определяет гибкость). Огромным преимуществом интегрированных сред (ИС) **Delphi** (http://www.borland.com.ru/delphi) и **C++Builder** (http://www.borland.com.ru/cbuilder) является значительная открытость системы - возможно добавление (и удаление) специфических объектов (в соответствии с требованиями конкретного разработчика - например, разработчика баз данных, специалиста по численному моделированию, создателя малтимедиа-приложений etc); в настоящее время существуют (часто бесплатно распространяемые) сотни мегабайт самых разнообразных объектов, разработанных третьими фирмами и/или независимыми разработчиками. Наличие компилятора с командной строкой в стиле MS-DOS, поставляемые в комплекте **EXPERT**'ы (аналоги **WIZARD**'ов фирмы Microsoft Corp.), регулярный выпуск новых версий **Delphi** делают **Delphi** не только удобным, но перспективным продуктом разработчика.

Недостатки **Delphi** - определенная 'раздутость' выполняемых файлов (всегда являющаяся оборотной стороной легкости этапа разработки); причем невозможность множественного наследования в базовом для **Delphi** языке программирования **Borland Object Pascal 8.0** практически не ограничивает разработчика.

Весной 1997 году Borland выпустила полностью **C**-базированную версии **RAD** / **VCL**-среды (получившую название **Borland C++Builder**), что позволило включить в список сегодняшних клиентов фирмы Borland армаду **C**-программистов. Есть надежды на достаточно быстрое и безболезненное внедрение **C++Builder**'а в практику разработчиков WINDOWS-приложений (хотя разработчики Borland не славятся эффективными **C**-компиляторами). Набор (палитра) компонентов **C++Builder**'а полностью соответствует таковой для **Delphi**. Последние версии указанных пакетов отличаются мощной под-

держкой разработки программных продуктов, предназначенных для эксплуатации в сети **InterNet** и объектов **ActiveX**.

В целом семейство интегрированных сред **Delphi** / **C++Builder** фирмы Borland столь хороши, что более чем конкурентоспособны с (несколько запоздавшим) пакетом **Developer Studio** фирмы Microsoft Corp. На грани нового 21 века Borland предложила аналог **Delphi** для разработки приложений на платформе Linux - интегрированную среду **Kylix** [14] (http://www.borland.ru/kylix/index.html), существенно расширяющую область применения ПО данной фирмы.

Разработчики пакетов **Delphi** и **C++Builder** расширяют ассортимент своих продуктов в иные области информационных технологий

- Поддержка Java-ориентированных технологий многоплатформенных приложений - собственноразработанная интегрированная среда **JBuilder** (http://www.borland.com.ru/jbuilder).
- Разработка средств управления и мониторинга объектных распределенных систем **Inprise AppCenter** (управление объектами технологии **CORBA** и **Enterprise Java Beans** (***EJB***), см. http://www.borland.com.ru/appcenter.
- Применение интегрированного комплекса средств **Inprise Application Server** (***IAS***) на основе стандартов **CORBA** и **J2EE** (http://www.borland.com.ru/appserver).
- Внедрения многозвенной архитектуры ***MIDAS*** (*MulIi-tier Distributed Application Services*); данная технология расширяет возможности предложенной ранее Microsoft Corp. технологии **DCOM** (*Distributed Component Object Model*) и поддерживается начиная с **Delphi** 3 и **C++Builder** 3 (http://www.borland.com.ru/midas).

Таким образом, в данной работе именно система **Delphi** выбрана базовой для разработки пользовательских WINDOWS-приложений; в соответствующих местах после заголовка абзаца **C++Builder** будут приведены данные по пакету **C++Builder**; подробнее о различиях см. также раздел 13.

На WEB-сайте http://imc.misa.ac.ru/pilger_roll автора данной работы можно найти примеры разработок на **Delphi** и **C++Builder**, иллюстрирующие некоторые особенности применения указанных ИС.

Введение в возможности **Delphi** и **C++Builder** по работе с базами данных приведено в методической разработке 'Введение в язык SQL запросов к базам данных' того же автора.

1. ЦЕЛЬ РАБОТЫ

Целью работы является дать первоначальные теоретические знания о

функционировании ОС WINDOWS и рассмотреть практические основы создания приложений в системах **Delphi** / **C++Builder**. После ознакомления с данным методическим руководством (желательно параллельно с работой на ПЭВМ) пользователь сможет разрабатывать простые WINDOWS-приложения; данная основа и постоянная практика позволят программисту стать профессионалом.

Для желающих самостоятельно изучать дополнительную литературу (а это наряду с ежедневной работой на ЭВМ является единственным способом приобрести профессиональные знания) можно рекомендовать литературные источники [1] для ознакомления с идеологией функционирования WINDOWS, [2] для знакомства с объектно-ориентированной версией языка **Pascal**, источник [3] для освоения малтимедийных возможностей WINDOWS, [4] для изучения функций WINDOWS API, [5,10] для создания приложений с помощью **Delphi**, [6,7,10] для освоения возможностей **Delphi** по работе с базами данных, [7] для практики работы с **Delphi** (в частности, разработки собственных компонентов), [8] как прекрасный справочник по стандартным компонентам **Delphi** и [9] в качестве руководства для продвинутых **Delphi**-программистов.

Для ознакомления с версией **C++** для **C++Builder**'а можно рекомендовать работу [11].

2. МОДЕЛЬ ПРОГРАММИРОВАНИЯ И КОНЦЕПЦИЯ МНОГОЗАДАЧНОСТИ В ОПЕРАЦИОННОЙ СИСТЕМЕ WINDOWS

Принципиальная разница между программированием на стандартном языке **C** и программированием в WINDOWS состоит в том, что все программы для WINDOWS разрабатываются на основе понятия передачи сообщений. Каждая программа для WINDOWS имеет цикл ввода сообщений и (для каждого из окон) - свою процедуру обработки сообщений.

Таким образом, сущность программирования в WINDOWS состоит в принятии сообщения (а при необходимости и генерации оного), пересылке его в то окно, для которого оно предназначено, возможно быстрой обработке этого сообщения и возврата к чтению следующего сообщения [1].

Зачем передавать сообщения ? Дело в том, что WINDOWS - многозадачная ОС, способная выполнять одновременно несколько программ; при этом ни одна программа не должна захватывать центральный процессор (ЦП) на слишком долгое время (так как это может вызвать ошибки в параллельно выполняемых и чувствительных к замедлению программах). В *приоритетных многозадачных* ОС (например, **UNIX**) выполнение программы прерывается по истечению априорно отведенного ей кванта времени (даже если в этот момент заканчивается важная и неотложная операция). В WINDOWS *парал-*

лельное выполнение не является приоритетным - передача управления от одной задачи к другой производится в явном виде самими этими задачами. WINDOWS функционирует как многозадачная ОС каждый раз, когда прикладная программа анализирует очередь сообщений. Если предназначенных данной программе сообщений не оказывается, WINDOWS начинает искать сообщения для других активных в данный момент программ и передает управление той, для которой сообщение имеется; далее процесс повторяется.

При разработке предназначенной для WINDOWS программы ее необходимо структурировать таким образом, чтобы она могла поддерживать передачу управления по вышеописанной схеме. Если не принять соответствующих мер, внешне корректная вновь созданная программа, возможно, будет работоспособной, но запущенные вместе с ней другие программы сильно рискуют остановиться вовсе (программа монопольно займет ресурсы всей ОС).

Заметим, что вышеприведенный метод реализации многозадачности в WINDOWS носит название *кооперативной многозадачности* (**cooperative multitasking**) и является единственным для WINDOWS'3.1; в ОС WINDOWS'9x (и, соответственно, WINDOWS'NT) реализован метод так называемой *вытесняющей многозадачности* (**preemptive multitasking**), при этом ОС сама решает, у какой программы отобрать управление и какой его передать (кооперативная многозадачность оставлена в WINDOWS'9x только в целях поддержки 16-битных приложений). Каждое 32-битное приложение в WINDOWS'9x имеет отдельную очередь сообщений и не зависит от того, как другие задачи работают со своими очередями (поэтому ситуация 'повисания системы' вследствие слишком редкой проверки очереди сообщений 16-битным приложением теоретически невозможна).

Кроме сказанного, 32-битные приложения в WINDOWS'9x могут использовать особый механизм - *многопоточность* (**multithreading**). В принятой для WINDOWS'9x терминологии исполняемое 32-битное приложение называется *процессом* (**process**); *процесс* состоит как минимум из одного *потока*.

Поток (**thread**) - часть кода программы, которая может выполняться одновременно с другими частями кода; *потоки* в пределах одного *процесса* используют одно адресное пространство, описатели оъъектов и иные общие ресурсы, но имеют отдельные счетчики команд; приоритетностью выполнения потоков (как и процессов) управляет ядро ОС на основе системы приоритетов. Потоки являются средством обеспечения параллельности обработки данных и часто используются для выполнения в программе *асинхронных операций* (**asynchronous operations**) - операций, которые могут быть инициализированы в любое время безотносительно к основному течению программы (лежащий на поверхности пример - периодическое сохранение редактируемого документа или опрос устройства пользовательского ввода - например, ‘мыши’ или клавиатуры). Потоки требуют меньших издержек, создаются (и

уничтожаются) быстрее процессов (поэтому их иногда называют ‘легковесными процессами’). В связи с использованием всеми потоками данного процесса общей памяти (конечно, за исключением индивидуальных стеков и содержимого регистров) обмен данными между ними предельно прост; по этой же причине создание использующего многопоточность приложения требует особой тщательности вследствие возможности случайного ‘пересечения’ по адресному пространству и/или нарушения последовательности операций (например, чтения/записи). ИС **Delphi** и **C++Builder** предоставляют родительский класс (в смысле ООП) **TThread** для реализации потоков [10].

Операционная система с *симметричной мультипроцессорной обработкой* (**symmetric multiprocessing, SMP**); такая как WINDOWS’NT, может выполнять на любом процессоре как код пользователя, так и код собственно ОС; при превышении числом потоков числа процессоров поддерживается многозадачность путем разделения времени каждого процессора между всеми ожидающими потоками.

Разработчики WINDOWS’NT ввели понятие *вол**о**кон* (**fibers**); *волокн**о**м* называется небольшой (‘облегченный’) поток, планировку которого осуществляет приложение.

2.1. **WINDOWS - ОПЕРАЦИОННАЯ СИСТЕМА, УПРАВЛЯЕМАЯ СОБЫТИЯМИ**

WINDOWS генерирует сообщение, когда происходит какое-либо событие или должно быть выполнено действие. Например, при перемещении 'мыши' (а это типичное событие) генерируется сообщение, указывающее (в числе других параметров) координаты точки, в которой находится курсор. Таким же образом сама ОС WINDOWS информирует прикладную программу о том, что в меню был выбран определенный пункт (например, 'Прекратить выполнение').

2.2. **ПОНЯТИЕ О СООБЩЕНИЯХ WINDOWS И ИХ ОБРАБОТКЕ**

Все сообщения в WINDOWS строятся в соответствии со строго определенным и компактным форматом. Любители **C++** могут заглянуть в файл **WINDOWS.H** и проанализировать структуру сообщений **MSG**

```
/* структура сообщения WINDOWS */
typedef struct tag MSG
{
   HWND hwnd;
   WORD message;
   WORD wParam;
   LONG  lParam;
```

```
   DWORD time;
   POINT pt;
}  MSG;
```

Первый элемент приведенной структуры указывает на окно, которому предназначено сообщение (каждому окну присвоен уникальный номер - **handle**, идентифицирующий окно в течение сеанса работы в WINDOWS). Второй элемент сообщения - идентификатор сообщения (начинающийся с символов **WM_** идентификатор, см. файл **WINDOWS.H**). Третий и четвертый элементы (**wParam** и **IParam**) несут дополнительную информацию. В элементе **time** содержится время помещения события в очередь, в **pt** - координаты 'мыши' в этот момент.

Если сообщение порождено входным событием, WINDOWS помещает его в общую для всех прикладных программ системную очередь сообщений. Помимо этого каждая прикладная программа имеет свою собственную очередь сообщений, куда помещаются (минуя системную очередь) сообщения, прямо адресованные конкретной программе.

Сообщения анализируются в цикле, который имеется в каждой написанной для WINDOWS программе. Ниже даны **C**- и **Pascal**-варианты указанного цикла.

```
//  C - вариант цикла обработки сообщений
MSG msg;  // объявление структуры типа MSG
while (GetMessage(&msg, NULL, 0, 0))
{
  TranslateMessage(&msg);
  DispatchMessage(&msg);
}

{ Pascal - вариант цикла обработки сообщений  }
while GetMessage(Message, 0, 0, 0) do
begin
  TranslateMessage(Message);
  DispatchMessage(Message);
end;
```

Функция **GetMessage** ищет (сканируя очередь) сообщение в очереди данной программы. Если сообщений не найдено, анализируется системная очередь, где распознаются поступившие от клавиатуры или 'мыши' сообщения; если таковых не оказалось, прикладная программа 'засыпает'. Функция **GetMessage** приостанавливает выполнение программы до тех пор, пока в очередь сообщений данной программы поступит какое-либо сообщение, далее поступившее первым сообщение извлекается и выполнение программы возобновляется.

Функция **GetMessage** возвращает значение **FALSE**, если получено сооб-

щение **WM_QUIT**, означающее, что программа должна закончить работу.

Функция **TranslateMessage** обрабатывает только поступившие с клавиатуры сообщения, преобразуя последовательность событий типа 'клавиша отпущена / клавиша нажата' в одно из следующих сообщений **WM_DEADCHAR**, **WM_SYSCHAR** и **WM_SYSDEADCHAR**.

Последняя в цикле функция **DispatchMessage** передает сообщение в окно (реально - связанной с данным окном процедуре обработки сообщений).

В действительности с каждым окном, создаваемым под управлением WINDOWS, связана некоторая функция обработки сообщений, именуемая оконной функцией **WinProc**. Связывание соответствующей **WinProc**-функции с заданным окном осуществляется присвоением имени **WinProc**-функции полю **lpfnWndProc** структуры типа **WNDCLASS**, определяющей данное окно, и регистрацией класса окна функцией **RegisterClass**.

При вызове **SendProc** или **DispatchMessage** в действительности вызывается именно функция **WinProc** (ниже приведен пример данной функции, оформленной как **C**-функция с **Pascal**-вызовом); приведенная функция обрабатывает всего два стандартных сообщения WINDOWS - **WM_PAINT** (перерисовать окно - в данном простейшем случае вывести в окно заданный текст) и **WM_DESTROY** (уничтожить окно)

```
LONG  FAR   PASCAL
WinProg_1(HWND hWnd,  // идентификатор данного окна
              WORD msg,    // код кообщения
              WORD wParam,
              LONG lParam)
{
 HDC hdc;  // дескриптор устройства вывода
 PAINTSTRUCT ps;  // параметры отображения окна
 RECT rect;  // размер клиентской области вывода

   switch (msg)
   {
    // . . . . . .
    case WM_PAINT:  // полность перерисовать окно
    hdc=BeginPaint(hWnd,  &ps);   // взять дескриптор устройства
    GetClientRect(hwnd, (LPRECT) &rect);   // определить область вывода
    DrawText(hdc, (LPSTR) ''Hello, BAKANOV !'',  // выдать текст
                -1, (LPRECT) &rect,
                DT_SINGLELINE | DT_CENTER | DT_VCENTER);
    EndPaint(hWnd, &ps);  // освободить дескриптор устройства
    break;
    // . . . . . .
    case WM_DESTROY:  // уничтожить окно
    PostQuitMessage(0);
    break;
```

```
    // обработать другие сообщения Windows

    default:  // обработчик по умолчанию (обязательная часть)
    return DefWindowProc(hWnd, msg, wParam, lParam);
    }

    return  0L;

} // конец процедуры
```

Выше было сказано, что каждое WINDOWS-окно имеет (связанную с ним) функцию **WinProc**. **WinProc**-функции часто имеют весьма громоздкий вид вследствие перенасыщенности операторами **switch/case** (иногда вложенными), что затрудняет непосредственную отладку программ. Однако тел**а** **case**-вариантов в различных (соответствующих разным окнам приложения) **WinProc**-функциях в основном повторяются. С переходом на **ООП**-технологию положение сильно упростилось - пишется одна (родительская) базовая функция **WinProc**, от которой наследуются (с модификацией - обычно заключающейся в дополнении возможностей) соответствующие процедуры для каждого окна.

Важно отметить, что позиция **default** в списке оператора **switch** обязательна (при вызове функции **DefWindowProc** происходит обработка сообщений, не учтенных соответствующими **case**-вариантами; в большинстве случаев процедура **DefWindowProc** является пустой - т.е. все неучтенные сообщения ‘теряются’).

2.3. ДВА СПОСОБА ПЕРЕДАЧИ СООБЩЕНИЙ

Сообщения может генерировать не только сама система WINDOWS, но и любая из поддерживаемых ею программ (причем отправить сообщение можно как самой себе, так и любой другой активной программе). Для передачи сообщений в окно существуют два различных способа - непрямой (он же отсроченный) и прямой (реализуемые WINDOWS API-функциями **PostMessage** и **SendMessage** соответственно, прот**о**типы этих функций в **C**-транскрипции приведены ниже).

```
BOOL
PostMessage(HWND  hwnd,   // дескриптор окна, которому
                          //  передается сообщение
            UINT  Msg,  //  собственно сообщение
            WPARAM  wParam,  //  первый параметр сообщения
            LPARAM  lParam);  //  второй параметр сообщения
```

Здесь **hWnd** - идентификатор (дескриптор) окна, которому передается со-

общение. При равенстве этого параметра **HWND_BROADCAST** сообщение передается всем окнам верхнего уровня в системе (включая недоступные, невидимые, перекрытые другими и всплывающие), за исключением дочерних окон. Если этот параметр равен **NULL**, то сообщение ставится в очередь (если она есть) сообщений текущего процесса. Параметр **Msg** определяет собственно передаваемое сообщение, параметры **wParam** и **lParam** содержат дополнительную информацию (при необходимости). При успехе функция **PostMessage** возвращает ненулевое значение, при неудаче - нуль (причину ошибки можно установить вызовом функции **GetLastError**).

```
LPRESULT
SendMessage(HWND  hwnd,   // дескриптор окна, которому
                          //  передается сообщение
            UINT  Msg,  //  собственно сообщение
            WPARAM  wParam,  //  первый параметр сообщения
            LPARAM  lParam);  //  второй параметр сообщения
```

Параметры функции **SendMessage** в основном идентичны таковым функции **PostMessage**; возвращаемое функцией значение зависит от вида сообщений.

При непрямой передаче сообщение помещается в очередь окна-адресата; если очередь не пуста, окно получит данное сообщение лишь после обработки всех предыдущих (на что потребно некоторое время).

При прямой передаче происходит обращение непосредственно к процедуре окна, минуя очередь (применяется при необходимости немедленной реакции на сообщение). Например, следующее WINDOWS API-предписание посылает сообщение **EM_LINEFROMCHAR** компоненту (понятие компонента см. ниже) **Memo_1** с целью получения номера первого выделенного в **Memo_1** символа

```
var
  LineNumb: Longint; { номер начального символа
                       выделенного   участка текста в Memo_1 }
  LineNumb:=SendMessage(Memo_1.Handle,EM_LINEFROMCHAR,
                        Memo_1.SelStart, 0);
```

В системах **C++Builder** и **Delphi** определена (в качестве метода класса **TControl**) функция **Perform**, обладающей функциональностью вышеприведенных

```
int __fastcall  Perform(Cardinal  Msg,
                        int WParam,
                        int LParam);
```

Например, нижеследующий **C**-оператор передает сообщение **WM_CLOSE** форме **Form_1**

```
Form_1 -> Perform(WM_CLOSE,  0, 0);
```

Пользователь имеет возможность определить (в дополнение к предоставляемым WINDOWS) и свои собственные сообщения - значение соответствующего идентификатора должно быть больше величины **WM_USER**. Например, можно определить пользовательское сообщение **WM_BAKANOV_IS_NOT_VERY_STUPID** в стиле **C** как

```
#define   WM_BAKANOV_IS_NOT_VERY_STUPID   (WM_USER+Ox13)
```

Константа **WM_USER** используется для разграничения зарезервированных для WINDOWS номеров сообщений и частных (определенных пользователем) сообщений. Все возможные номера сообщений разделены на 5 диапазонов

от **0** до **WM_USER-1**	Номера сообщений, используемые WINDOWS
от **WM_USER** до **0x7FFF**	Номера частных сообщений внутри данного класса оконных компонентов
от **0x8000** до **0xBFFF**	Зарезервированные для будущего использования в WINDOWS номера сообщений
от **0xC000** до **0xFFFF**	Номера, соответствующие строкам сообщений, используемых для обмена между приложениями и зарегистрированным функцией **RegisterWindowMessage** окном
выше **0xFFF**	Зарезервированные для будущего использования в WINDOWS номера сообщений

Для объявления пользовательского сообщения следует определить его имя

```
#define WM_MyMessage  WM_USER
```

и в необходимый момент послать это сообщение

```
SendMessage(Form1->Handle, WM_MyMessage, 0, iMessage);
```

здесь **iMessage** - целое число, являющееся параметром сообщения. Для обработки сообщения в теле адресата должна быть объявлена и реализована соответствующая функция обработки (иначе сообщение будет принято, но не обработано - будет вызван обработчик по умолчанию).

Синтаксис **C** / **C++** требует наличия подобного нижеприведенному ис-

ходного кода (в базовом для **Delphi** языке **Object Pascal** подобная функциональность реализуется значительно проще, см. ниже)

```
//  модуль  U1_Mess_2.h
. . . . . .
#define  WM_MyMessage  WM_USER
. . . . . .
class TForm1: public TForm
{
  . . . . .
  private:  //  User declarations
     void __fastcall OnMyPost(TMessage &Message);
  public:  // User declarations
     __fastcall TForm(Tcomponent *Owner);
    BEGIN_MESSAGE_MAP
    . . . . .
     MESSAGE_HANDLER(WM_MyMessage, TMessage,  OnMyPost)
    END_MESSAGE_MAP(TComponent)
};
// ---------------------------------------------------------------------------

//  модуль  U1_Mess_2.cpp
. . . . .
void __fastcall  TForm1::OnMyPost(TMessage &mess)
{
  Label1->Caption = ''Получено сообщение '' + IntToStr(mess.LParam);
}
```

Таким образом, обработка сообщений в WINDOWS происходит по следующей цепочке

событие → **MainWndProc** → **WndProc** → **Dispatch** → ***обработчик события***

Например, С-оператор

```
PostMessage(Form2->Handle, WM_CLOSE, 0, 0);
```

передает окну формы **Form2** сообщение **WM_CLOSE**, закрывающее это окно. Оператор

```
PostMessage(FindWindow("TForm1", "Приложение_Primer_2",
                WM_CLOSE, 0, 0);
```

передает аналогичное сообщение окну класса **TForm1** с заголовком **Приложение_Primer_2** (используя WINDOWS API - функцию **FindWindow**).

Например, нижеприведенный вызов

```
SendMessage(Form2->Handle, WM_CLOSE, 0, 0);
```

передает окну формы **Form2** закрывающее это окно сообщение **WM_CLOSE**. Вызов

```
SendMessage(FindWindow("TForm1", "Приложение_Primer_2",
                WM_CLOSE, 0, 0);
```

передает аналогичное сообщение окну класса **TForm1**.

Функция **PostMessage** возвращает управление вызвавшей ее программе, не дожидаясь окончания обработки сообщения (этим она существенно отличается от функции **SendMessage**, которая возвращает управление вызывающей программе только после окончания обработки сообщения и на это время блокирует пославшее сообщение приложение).

Вышеприведенные данные дают некоторые основы знания о функционировании ОС WINDOWS, в то же время демонстрируя сложность предмета (для заинтересованных в профессиональных знаниях рекомендуется работы [1÷2,4÷11]). Для работающих на **MFCL** / **OWL** **C++** знание всех тонкостей необходимо, начинающий **Delphi**-разработчик может и не иметь представления о тонкостях функционирования WINDOWS-программ (хотя серьезная дальнейшая работа в WINDOWS и потребует дополнительных знаний).

Например, в **Delphi** доступно прямое связывание сообщений WINDOWS с *методами-обработчиками* (**message-handling methods**). Прототип соответствующей процедуры-обработчика должен быть объявлен с директивой **message**, непосредственно за которой следует идентификатор обрабатываемого сообщения

```
procedure WM_Reaction1 (var Message: TWMSize); message
                                                WM_MOUSEMOVE;
```

Теперь при генерации сообщения **WM_MOUSEMOVE** (любое перемещение 'мыши') будет вызвана процедура **WM_Reaction_1** (название метода, а также имя и тип единственного описанного с квалификатором **var** формального параметра в данном случае не имеют значения)

```
procedure TForm1.WM_Reaction_1(var Message: TWMSize);
begin
  Label_1.Caption:='Получено очередное сообщение ' +
                   'о перемещении мыши...';
end;
```

Приведенный подход доступен для квалифицированных разработчиков, в

большинстве же случаев **Delphi** / **C++Builder** - пользователь даже не подозревает о существовании сообщений имея дело только с определенными событиями.

3. **ИНТЕГРИРОВАННЫЕ СРЕДЫ Delphi / C++Builder**

Стартовый файл интегрированной среды (ИС) **Delphi** называется **DELPHI32.EXE** (для **C++Builder**'а файл **BCB.EXE**) и стартуется стандартными средствами WINDOWS.

При старте ИС создает главное окно, 'нависающее' над текущими окнами WINDOWS (см. рис.1).

На приведенной копии экрана ПЭВМ окно ИС **Delphi 4** расположено в виде полоски в верхней части экрана, пользователю доступно горизонтальное меню и линейка кнопок 'быстрого' вызова команд, дублирующая команды главного меню.

Кроме непосредственно ИС, в стартовом окне WINDOWS доступны другие компоненты системы **Delphi** - утилиты, вспомогательные модули и др. ИС **Delphi** включает все основные компоненты интегрированных сред WINDOWS - текстовый редактор, редактор ресурсов, компилятор, встроенный отладчик, систему настройки самой ИС, систему контекстной помощи и серию вспомогательных утилит.

Внешний вид окон **C++Builder**'а практически полностью соответствует таковым для **Delphi**, принципы работы в ИС также подобны. В дальнейшем для сокращения термин '**Delphi**' будет применяться к обоим интегрированным средам (**Delphi** и **C++Builder**), если разница между обоими ИС в данном контексте несущественна.

Рис.1. Копия экрана дисплея ПЭВМ при загруженной ИС **Delphi**

3.1. **СЧИТЫВАНИЕ, СОХРАНЕНИЕ И СОЗДАНИЕ ПРОЕКТОВ**

Считывание (существующего) проекта доступно через главное меню (см. рис.1) **File|Open Project**, запись (сохранение) - **File|Save Project** (здесь и далее вертикальная черта отделяет выбор из горизонтального меню и далее выбор из открывающегося подменю). Изменение имени проекта доступно сохранением его через вариант **File|Save Project As...**, закрытие проекта (очистка рабочей области ИС) - **File|Close Project**.

Создание нового проекта инициализируется **File|New Project**. Новый проект по умолчанию получит имя **Project1**; **Delphi** / **C++Builder** потребует введения реального имени проекта (как и имени каждого модуля) в момент сохранения проекта.

Выход из ИС реализуется выбором **File|Exit**.

3.2.**ОСНОВНЫЕ ФАЙЛЫ ПРОЕКТА DELPHI**

Каждый проект (фактически список необходимых при создании конкретного пользовательского приложения данных - исходных файлов, параметров компиляции etc) состоит из нескольких (иногда нескольких десятков) файлов,

причем каждый из них необходим.

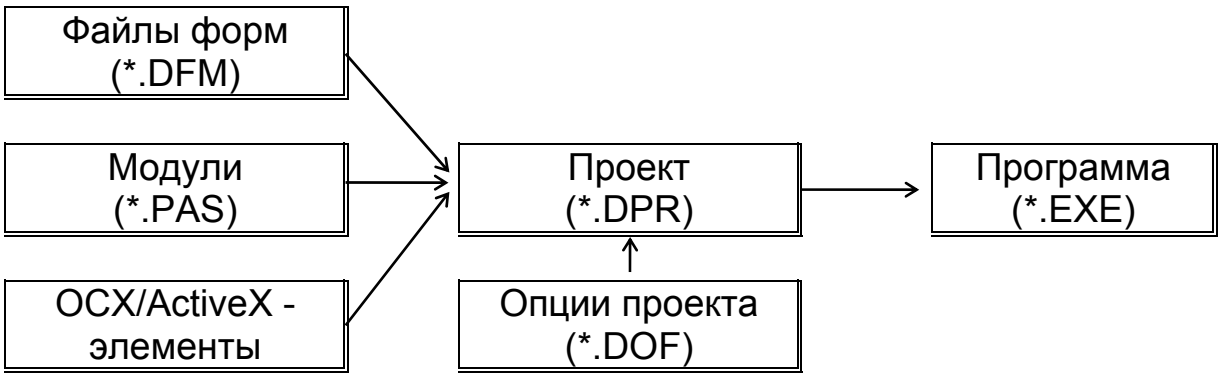
Настоятельно рекомендуется для каждого проекта выделять отдельный каталог ! Отказ от этого правила неминуемо приведет к полной дезорганизации последующей работы (некоторые файлы разных проектов имеют по умолчанию одинаковые имена).

Главный файл проекта имеет расширение **DPR** (*Delphi Project*, для **C++Builder** расширение **BPR**), совпадающее с именем проекта имя и содержит список всех необходимых для проекта файлов.

Для каждого окна (формы в терминологии **Delphi**) создаются минимум два файла - текстовый файл с расширением **PAS** (содержащий исходный текст модуля на **Pascal**'е) и двоичный файл с расширением **DFM** (содержащий иерархическое представление содержащихся в модуле компонент **Delphi**). Заметим, что **DFM**-файл может быть просмотрен (и отредактирован - чего, кстати, не следует делать новичкам) в ИС в текстовом виде (как, впрочем, и любой текстовый файл) посредством выполнения **File|0pen File**, **File|Save File**, **File|Save File As...** и **File|Close File**.

Проект включает в себя (текстовый) файл опций проекта (расширение - **DOF**) и, возможно, файл (файлы) формата **VBX** (*Visual Basic eXtensions,* начиная с **Delphi 2.0** файлы **VBX** не используются, их роль выполняют **OCX/ActiveX**-компоненты).

Ниже приведена диаграмма, иллюстрирующая процесс создания исполняемого файла в системе **Delphi**.



Для системы **C++Builder** объединяющий файл имеет расширение **BPR**, в проекте присутствуют **DFM**, **CPP**, **H**, **RES** - файлы. Общего правила '*отдельный проект = отдельный каталог*' рекомендуется придерживаться неизменно. Вышеприведенный список файлов проекта минимален - в процессе работы над проектом могут требоваться **BMP**, **DCR**, **ICO**, **DSK**, **HLP** и другие файлы.

При перенесении проекта между ПЭВМ следует транспортировать все (за исключением включающих в расширение тильду - волнистую черту - файлов). Заметим, что применяющий компоненты третьих фирм разработчик должен перенести также файлы указанных компонентов и инсталлировать их

(при этом модифицируются файлы библиотеки).

C++Builder. В этом случае прототипы функций и описания данных помещаются в файл с расширением **H** (который, впрочем, также автоматически модифицируется при проектировании приложения). **H**-файл фактически заменяет USES'ы в стиле **Delphi**.

4. СОЗДАНИЕ ПРОСТЫХ WINDOWS-ПРИЛОЖЕНИЙ

Создание приложений (пользовательских программ) в среде **Delphi** не требует знания тысяч тонкостей программирования под WINDOWS (хотя по мере повышения требований и собственной квалификации пользующийся **Delphi** разработчик неизменно вынужден будет постепенно постигать тонкости WINDOWS-программирования).

Delphi / **C++Builder** скрывают (инкапсулируют) от пользователя бесконечные тонкости WINDOWS-программирования, позволяя строить приложения 'на лету' - буквально в течении минут создавая готовое WINDOWS-приложение.

4.1. ФОРМА - ОСНОВА РАЗРАБОТКИ ПРИЛОЖЕНИЯ В Delphi

Основным интерфейсным элементом в **Delphi** является форма (**Form**). С точки зрения WINDOWS форма суть прототип будущего окна WINDOWS. С точки зрения **Delphi** форма представляет собой присущий любой исполняемой с этой среде программе визуальный компонент, являющийся контейнером (т.е. включающий в себя) другие компоненты, определяющие функциональность данного приложения.

Заметим, что приложение может содержать несколько форм (многооконное приложение), несущих каждая свою функциональную нагрузку и активизируемых/закрываемых в нужный момент.

На рис.1 приведена копия экрана дисплея, содержащая (в правой части) пустую форму с именем **Form1** (форма является принадлежащим к классу **TForm Delphi** - компонентом). Созданная в проекте первой форма будет являться главной (появляющейся первой) формой приложения.

В принципе содержащий одну форму (и только форму) проект вполне работоспособен и может быть откомпилирован и выполнен (для компиляции следует использовать **Compile|Compile** или сочетание клавиш **Ctrl+F9** или **Compile|Build All**, для запуска на исполнение **Run|Run** или **F9**). При старте такого EXE-файла на экране дисплея появится пустое, не реагирующая на нажатия клавиш WINDOWS-окно, которое, однако можно перемещать по экрану, изменять его размеры, минимизировать, свертывать в иконку и закрывать. Заметим, при этом пользователем не написано ни строки кода !

В левой части рис.1 видно окно **Object Inspector**'а (Инспектора Объектов)

- инструмента, с помощью которого пользователь устанавливает свойства объектов и назначает (во время разработки программы - **DesignTime**) методы-обработчики событий. На примере использования **Object Inspector**'а при проектировании формы рассмотрим особенности его функционирования при конкретизации свойств и методов всех других объектов (**Object Inspector** вызывается **View|Object Inspector** или клавишей **F11**).

В верхней части окна **Object Inspector**'а имеется выпадающий список (фактическое выпадение происходит при одинарном щелчке 'мыши' по расположенной справа вверху кнопке со стрелкой вниз) включенных в данную форму объектов - компонентов **Delphi** (включая саму форму).

В нижней части окна **Object Inspector** находятся два ярлычка - переключателя между страницами **Properties** (свойства) и **Events** (сообщения), см. рис.2,3; переключения между ними осуществляется одинарным щелчком 'мыши' по соответствующему ярлычку.

Каждая строка окон **Properties** и **Events** соответствует конкретному свойству/сообщению соответственно; причем левый (неизменяемый пользователем) столбец содержит имена свойств/сообщений, а правый - задаваемое пользователем значения свойства (для окна **Properties**) или имени вызываемой при генерации выбранного сообщения процедуры (для окна **Events**).

Заметим, что в большинстве случаев пользователю нет необходимости вводить какие-либо значения - **Delphi** предлагает их величины и названия по умолчанию.

4.1.1. **НАСТРОЙКА СВОЙСТВ ФОРМЫ**

Рассмотрим процесс задания свойств формы (см. рис.2,3); упомянув только важнейшие из них (пользователь всегда может использовать систему контекстной помощи или литературные источники).

Свойство **BorderIcons** определяет наличие трех стандартных для WINDOWS кнопок данной формы - **biSystemMenu**, **biMinimize** и **biMaximizee**. Редактирование этих свойств производится выделением строки **BorderIcons** в окне **Properties** Инспектора Объектов (выделенная строка изменяет цвет), нажатием правой клавиши 'мыши' для появления всплывающего меню и выбора варианта **Expand** в этом меню (см. рис.1), после чего в строке **BorderIcons** расширится (признаком расширяемости строки служит знак плюс в качестве первого символа строки) до трех строк - **biSystemMenu**, **biMinimize** и **biMaximize**; затем появляется возможность установить каждое из этих свойств в **TRUE/FALSE** (одинарным щелчком 'мыши' на появившейся в правой части строки кнопке со стрелкой вниз и выбором соответствующего значения или просто двойным щелчком 'мыши' по соответствующей строке). ‘Свернуть’ свойства в строке можно выбором варианта **Collapse** во всплывающем по нажатию правой клавише 'мыши' меню.

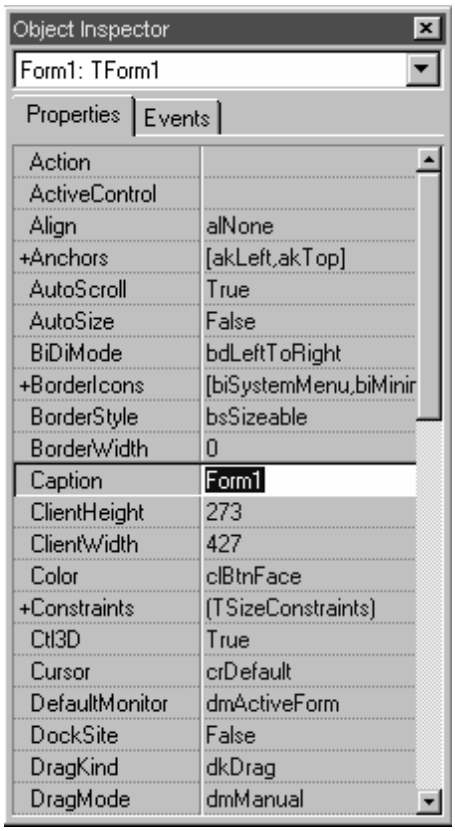


Рис.2. Окно **Object Inspector**'а в режиме редактирования свойств (**Properties**) выбранного компонента.

Тип рамки формы (окна) задается свойством **BorderStyle**. По умолчанию это значение установлено в **bsSizeable** - форма может изменять свой размер. Допустимо указание следующих значений - **bsDialog** (форма в виде диалоговой панели), **bsSingle** (тонкая рамка), **bsNone** (рамка отсутствует).

Свойство **Position** определяет местоположение формы (при ее первоначальном появлении на экране). По умолчанию используется **poDesigned** - форма располагается в том месте экрана и имеет те размеры, которые были заданы при ее создании в **DesignTime**. Возможны иные значения свойства **Position** - например, **poScreenCenter** (сохраняется размер **DesignTime**, но форма располагается в центре экрана) и др.

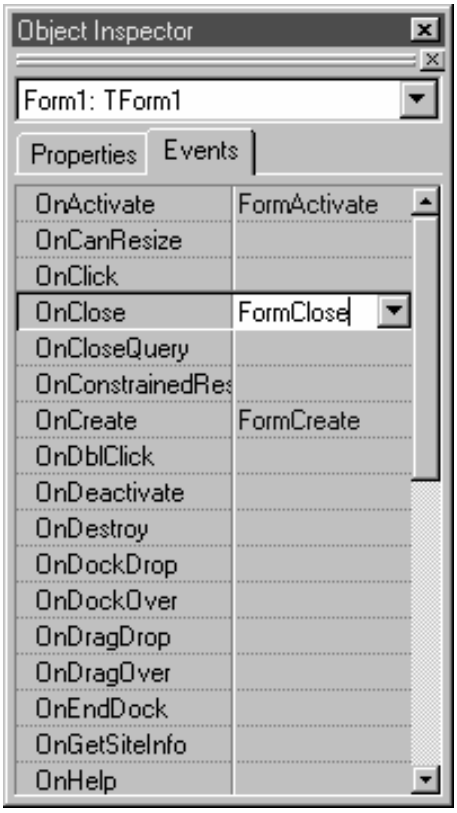


Рис.3. Окно **Object Inspector**'а в режиме назначения связей событий (**Events**) и методов-обработчиков данного события для выбранного компонента.

Свойство **WindowState** задает начальный способ отображения формы (окна). При **WindowState=wsNormal** (умолчание) форма отображается с размерами **DesignTime**, значения **wsMaximized** и **wsMinimized** соответствуют расширению формы во весь экран дисплея и свертке окна соответственно (свойство **Icon** задает иконку, отображаемую при минимизации формы).

Свойство **Menu** указывает на компонент типа **TMainMenu** (стандартная для WINDOWS полоса меню, включающее вложенные меню).

Свойство **PopUpMenu** указывает на компонент типа **TPopUpMenu** (вплывающее по нажатию правой клавиши 'мыши' меню).

Свойство **Name** задает пользовательское имя компонента (вместо **Form1**, **Form2**, **Form3** и т.д. по умолчанию), **Caption** - появляющийся в верхней час-

ти окна текст-заголовок окна, **Color** позволяет задать цвет формы, **Hint** - появляющуюся при небольшой задержке курсора 'мыши' над некоторым элементом интерфейса строку текста, являющуюся контекстной подсказкой (**Hint** является аббревиатурой словосочетания '*Help Instance*', необходимо установить в **TRUE** свойство **ShowHint**), **HelpContext** - задать номер темы помощи при вызове системы контекстной помощи (по клавише **F1** во время исполнения программы - **RunTime**, значение **HelpContext=0** отменяет вызов темы помощи для данного компонента), **Cursor** -определяет форму курсора 'мыши' в момент нахождения курсора в пределах данного компонента.

Свойство **FormStyle** формы определяет тип окна - простое (**SDI**-окно) или содержащее дочерние окна (**MDI**-окно, значение свойства для 'родительского' окна **fsMDIForm**, для 'дочернего' окна - **fsMDIChild**). Для функционирования окна 'поверх всех' необходимо установить свойство **FormStyle=fsStayOnTop**.

У каждого компонента **Delphi** свои свойства, для уяснения которых следует пользоваться системой контекстной помощи и/или внешней документацией (для компонентов третьих фирм).

Важно, что присваивать значения свойствам можно и во время выполнения приложения (**RunTime**), для этого выполняется простое присваивание типа нижеприведенного

```
Form1 .ShowHint: =TRUE;
Form1.Hint:='Это строка текста контекстной помощи';
```

C++Builder. Согласно синтаксису C++ вышеприведенный код должен быть записан как

```
Form1->ShowHint=true;  //  для C++  в 'true /  false' все буквы прописные !
Form1->Hint="Это строка текста контекстной помощи";
```

4.1.2. НАСТРОЙКА СВЯЗИ СОБЫТИЙ С ПРОЦЕДУРАМИ ИХ ОБРАБОТКИ

Установка связи событий с обрабатывающими процедурами также проста и производится на странице **Events** Инспектора Объектов (см. рис.3).

Для задания нужной процедуры следует выбрать строку соответствующего события и единожды щелкнуть кнопкой 'мыши' по кнопке со стрелкой вниз, появляющейся в правой части соответствующей строки (при этом будет показан список уже спроектированных процедур, соответствующих данному событию по списку формальных параметров; пользователю остается выбрать нужную) или дважды щелкнуть на соответствующей строке (в этом случае **Delphi** сгенерирует уникальное имя процедуры и создаст прототип и заготовку ('пустышку') этой процедуры с отсутствующим телом процедуры; имя

процедуры можно в дальнейшем изменить) или просто ввести желаемое имя процедуры (прототип и 'пустышка' также будут сгенерированы).

Например, на рис.3 процедуры **FormActivate**, **FormClose**, **FormCreate** и **FormResize** связаны (т.е. будут вызываться при возникновении соответствующих событий) с событиями **OnActivate** (возникает в момент активизации формы), **OnClose** (возникает в момент закрытия формы), **OnCreate** (возникает при создании формы) и **OnResize** (возникает при изменении размеров формы) соответственно.

Например, ниже приведена спроектированная **Delphi** заготовка ('пустышка') процедуры **FormActivate** (текст **TForm1.FormActivate** говорит о принадлежности процедуры **FormActivate** объекту типа **TForm1**)

```
procedure TForm1.FormActivate(Sender: TObject);
begin

end;
```

В описательную часть (после служебного слова **type** для **Delphi**) будет также помещен прототип процедуры в виде строки

```
procedure Form1.Activate(Sender: TObject);
```

C++Builder. В этом случае 'пустышка' (в файле ***.CPP**) и прототип (в файле ***.H**) имеют вид (квалификатор **__fastcall** задает режим передачи параметров функции не через стек, а через регистры процессора - если это возможно, конечно)

```
void  _fastcall  TForm1::FormActIvate(Tobject *Sender)
{

}

void _fastcall  Form1::Activate(TObject *Sender);
```

Текст (тело процедуры) между служебными словами **begin** и **end**; (открывающей и закрывающей фигурными скобками для **C++Builder**'а) заполняет пользователь, определяя тем самым функциональность данной процедуры. Естественно, возможно определение любого количества процедур, не связанных с событиями; каждая также требует прототипа.

Заметим, что формальный параметр **Sender** в заголовке функции указывает на объект, вызвавший событие.

Часто приходится строить процедуры обработки следующих событий - **OnClick** и **OnDbClick** (одинарный или двойной щелчок 'мышью'), **OnKeyDown**, **OnKeyPress** (нажатие и отпускание клавиши), **OnMouseDown**, **On-**

MouseUp, **OnMouseMove** (отпускание, нажатие клавиши и перемещение 'мыши'), **OnPaint** (требование перерисовки объекта).

Для реализации обработки нажатий клавиши методами формы следует установить **KeyPreview=TRUE** и создать процедуру-обработчик события **OnKeyDown**

```
Procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
                                    Shift: TShiftState);
Begin
  if Printer.Printlng AND { если происходит печать... И... }
     (Key=VK_ESCAPE) then { ...нажата клавиша ESC }
  begin
    Printer.Abort;  { остановить печать }
    MessageDlg('Вывод на печать прерван пользователем',
                    mtInformation, [mbOk], 0);
  end;
End;  { конец процедуры }
```

Важно, что связывать события с их обработчиками можно и во время выполнения приложения (**RunTime**), для этого выполняется простое присваивание типа нижеприведенного (в правой части выражения находится имя процедуры-обработчика события):

```
Form1.0nCreate:=FormCreate:
```

C++Builder. Соответственно

```
Form1->OnCreate = FormCreate;
```

Для переключения между окном формы и окном исходного текста (при создании тела процедур) служит клавиша **F12**. Другой способ - **View|Project Manager** (или сочетание клавиш **Ctrl+V+P**) и в дальнейшем выбор нужной формы из предлагаемого списка и использование кнопок **View unit** для показа исходного текста формы или **View form** для показа внешнего вида формы (см. рис.4).

Окно **Project Manager** позволяет добавлять и изымать модули и формы из проекта, а также устанавливать некоторые параметры проекта.

На рис.5 приведено окно **Project Options**, вызываемое кнопкой **Options** из **Project Manager** или путем выбора **Options|Forms** в главном меню. Здесь в левом подокне приведен список форм, создаваемых **Delphi** автоматически (в момент старта программы, именно так функционируют большинство форм), в правом подокне - список допустимых форм (создаваемых динамически во время работы **Delphi**-программы, что бывает необходимо в больших проектах). Динамически создаваемые формы используют компонентный метод

Create для создания и метод **Free** для разрушения; связанные вопросы сложны и данном руководстве не рассматриваются.

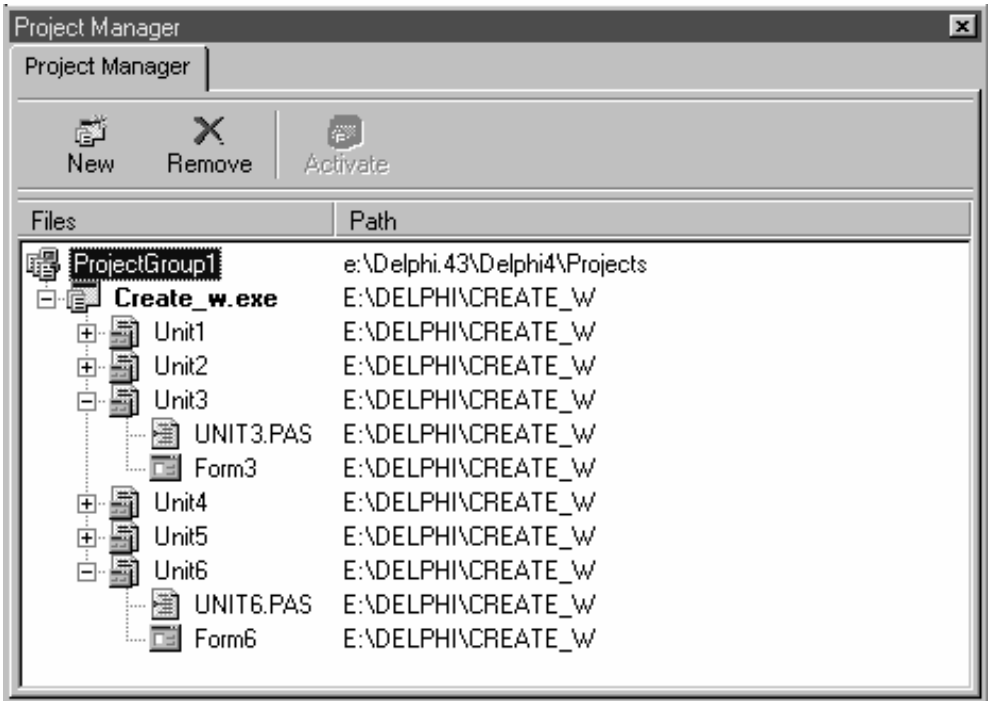


Рис.4. Окно **Project Manager** для управления модулями и формами

4.1.3. **УСТАНОВКА СВОЙСТВ ВО ВРЕМЯ ВЫПОЛНЕНИЯ ПРИЛОЖЕНИЯ**

Выше была рассмотрена методика настройки свойств компонентов **Delphi / C++Builder** во время проектирования приложения (**DesignTime**), однако большинство свойств может быть успешно установлено (переустановлено) в период выполнения EXE-модуля (**RunTime**). Целесообразно производить это в функции, связанной с событием **OnCreate** (или **OnActivate**) для формы (события при создании формы происходят в следующей последовательности - **Create**, **Activate**, **Paint**). В нижеприведенном примере связанная с событием **OnCreate** (создание формы) процедура **FormCreate** вызывает процедуру чтения файла конфигурации и устанавливает свойство **FormStyle** в значение **fsMaximized** ('растягивает' форму во весь экран), а вызываемая в момент активизации формы процедура **OnActivate** выполняет присваивание значения переменной **FlagPreview** свойству **Checked** компонента **Component_N7** и вызывает процедуру **LoadFileAnimateOfBanner** загрузки файла анимации флага.

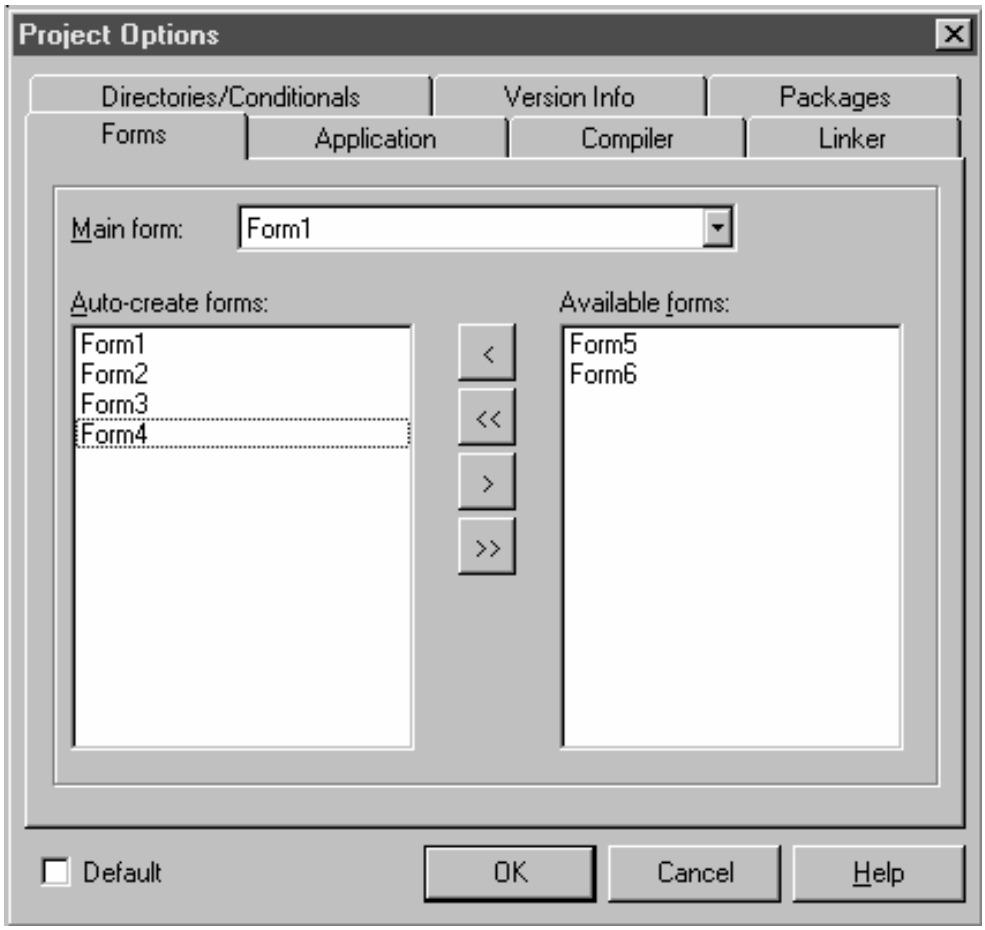


Рис.5. Окно **Project Options** управления параметрами проекта

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   ReadConfig;  { прочитать файл конфигурации }
   FormStyle=wsMaximized;
end; { конец процедуры }

procedure TForm1.OnActivate(Sender: TObject);
begin
   Component_N7.Checked := FlagPreview;
   LoadFileAnimateOfBanner;  { загрузить файл анимации флага  }
end; { Баканов В.М.,  МГАПИ,  кафедра ИТ-4, 1996-2000 }
```

Вышеприведенный механизм перестройки параметров компонентов придает дополнительную мощность и гибкость спроектированным с помощью **Delphi** приложениям.

4.1.4. **МОДАЛЬНЫЙ И НЕМОДАЛЬНЫЙ ДИАЛОГИ**

Реальный проект состоит из многих (часто десятков) форм (окон), каждое

из которых активируется в виде реакции на некоторое событие (нажатие кнопки 'мышью' в простейшем случае).

Существует два основных типов диалога - *модальный* (активное в данный момент окно перехватывает все сообщения и до его закрытия обращение к другим окнам - даже присутствующим на экране - невозможно) и *немодальный* (возможна активизация - например, щелчком 'мыши' - любого из присутствующих на экране окон). Наиболее часто используются модальные диалоги (например, подобные WINDOWS'95 системы использует в основном модальные диалоги), однако некоторые приложения (например, известный из WINDOWS'3.1x модуль SYSEDIT) строятся на основе немодальных диалогов.

Модальный вызов формы оформляется с использованием компонентного метода **ShowModal** (в нижеследующей строке модально вызывается форма с именем **FormMediaData**)

```
FormMediaData.ShowModal;
```

Немодальный вызов той же формы производит метод **Show**

```
FormMediaData.Show;
```

C++Builder. Соответственно

```
FormMediaData->ShowModal();
FormMediaData->Show();
```

В обоих случаях метод **Close** закрывает форму.

Немодальные диалоги представляет при проектировании и использовании определенные сложности, начинающим разработчикам вряд ли следует применять их без крайней на то необходимости.

4.1.5. СТАНДАРТНЫЕ ФОРМЫ-ПАНЕЛИ СООБЩЕНИЙ

Система **Delphi** предоставляет пользователю (заранее предопределенные) формы - окна диалоговых панелей сообщений.

Функция **MessageDlg** позволяет вывести сообщение пользователю и имеет возможность включить в форму несколько кнопок для выбора ответа пользователя. Например, нижеследующий пример стандартного использования функции **MessageDlg** для закрытия программы при вызове выводит соответствующий текст в форме с графическим символом подтверждения (согласно константе **mtConfirmation**) и две кнопки с надписью **Ok** и **Cancel** (константы **mbOk** и **mbCancel**), используя тему контекстного HELP'a с номером 9996 и производит действия в соответствии с возвращаемым **MessageDlg** значением

```
Procedure TForml.FormClose(Sender: TObject;
                                  var Action: TCloseAction);
Begin { запросить подтверждение на закрытие формы }
   if MessageDlg('Вы в самом деле хотите закончить работу ?',
                    mtConfirmation, [mbYes, mbNo],
                    9996) = mrYes  { была нажата кнопка Yes }
   then
     Action := caFree { была нажата клавиша кнопка Yes - закрыть форму }
   else
     Action := caNone; { кнопка No - игнорировать закрытие формы }
End;  {конец процедуры}
```

C++Builder. В этом случае следует воспользоваться конструкцией

```
void _fastcall TForm1::FormClose(TObject *Sender,
                                       TCloseAction &Action)
{
  switch (MessageBox(0,"Вы в самом деле хотите закончить работу ?",
                           "Предупреждение....",
                            MB_YESNO | MB_ICONWARNING | MB_TOPMOST))
   {
     case IDYES: Action=caFree; // нажата кнопка Yes
                 break;
     case IDNO:  Action=caNone; // нажата кнопка No
                 break;

   } //конец блока SWITCH
} // конец функции FormClose

// допустимы также вызовы Delphi-подобных функций диалогов
// для задания кнопок на форме используются битовые поля -
// пример использования этих полей см. ниже
// if (MessageDlg('Вы в самом деле хотите закончить работу ?",
//                    mtConfirmation,
//                   TMsgDlgButtons() «  mbYes «  mbNo,
//                    9996) = IDYES)
//  и так далее...
```

Функция **MessageDlgPos**, кроме прочих, содержит параметр, позволяющий указать положение формы на экране (**MessageDlg** всегда выводится в центре экрана).

Функция **InputBox** используется для вывода формы, содержащей строку ввода и две кнопки - **Ok** и **Cancel**. Функция возвращает либо введенную пользователем строку, либо описанную параметром **ADefault** строку.

Функция **InputQuery** возвращает введенную пользователем строку или строку **ADefault** при нажатии любой кнопки. Введенная пользователем стро-

ка возвращается в параметре **Value**, возвращаемое функцией значение есть **TRUE** при нажатии кнопки **Ok** или **FALSE** при нажатии **Cancel** или **Esc**.

4.1.6. СТАТИЧЕСКОЕ И ДИНАМИЧЕСКОЕ ИСПОЛЬЗОВАНИЕ КОМПОНЕНТОВ

Являющаяся компонентом форма может быть использована статически; в этом случае в **DPR**-файле присутствует строка типа

```
Application.CreateForm(TForm1, Form1);
```

говорящая о том, что форма **Form1** автоматически создается в момент начала выполнения приложения (и обычно существует до окончания работы оного). В этом случае **Form1** присутствует в левой части (**Auto-Create Forms**) окна **Project Options** (закладка **Forms**, полный доступ к окну суть **Options|Project|Forms**).

Однако форма может создаваться динамически в нужный момент и соответственно уничтожаться за ненадобностью; в некоторых случаях при этом удается добиться снижения общего объема требуемой для работы приложения памяти. Имя такой (динамической) формы должно быть занесено в правую часть (**Available Forms**) вышеуказанного окна **Project Options**, а сама форма должна в нужном месте создаваться компонентным методом (конструктором) **Create**, а уничтожаться с использованием деструктора **Free** (или **Destroy**).

Все сказанное относится к любому компоненту и объекту **Delphi** - ссылка на объект сначала должна появиться в описательной части (после ключевого слова **var**), а затем сам объект (со всеми дочерними компонентами, для которых он является контейнером) физически создается с помощью метода **Create**.

5. ЧАСТО ИСПОЛЬЗУЕМЫЕ ЭЛЕМЕНТЫ WINDOWS И ИХ ПРИМЕНЕНИЕ

Как было сказано, форма является также контейнером для обеспечивающих пользовательский интерфейс компонентов (**Control**'ов в терминологии WINDOWS-программистов). Именно с помощью этих компонентов пользователь осуществляет ввод данных, управление режимами выполнения программы и анализирует полученные результаты.

Компоненты выбираются разработчиком из расположенной в правой верхней части окна **Delphi** палитры компонентов (рис.1,6) нажатием левой клавиши 'мыши' и перетаскиваются в нужное место на форме; в дальнейшем уточняется их положение на форме и происходит настройка с помощью вы-

зова **Object Inspector**'а для каждого конкретного компонента (нужный компонент выбирается 'мышью', признаком выбранного компонента служит рамочка из черных квадратиков вокруг компонента).

В дальнейшем рассмотрим наиболее часто используемые интерфейсные компоненты **Delphi** (названия их обычно соответствуют стандартной терминологии WINDOWS-интерфейса); весьма полное описание компонентов **Delphi** (приведены свойства, методы и сообщения) содержится в книге [8].

На рис.6 приведена копия экрана дисплея в период проектирования приложения (**DesignTime**); показано окно ИС **Delphi** / **C++Builder** (в палитре компонентов выбрана страница **Standard**), форма **Form1** с размещенными на ней компонентами.

Для уточнения расположения компонентов на форме следует выбрать нужный компонент (одинарным щелчком 'мыши' в пределах компонента, выбранный компонент выделяется рамкой из черных квадратиков, см. второй сверху компонент во втором слева ряду компонентов на рис.6) и перемещать его по форме, держа нажатой левую клавишу 'мыши'. Точное позиционирование можно проводить, используя сочетание клавиш **Ctrl**+стрелки для перемещения компонента как целое и **Shift**+стрелки для изменения размеров компонента.

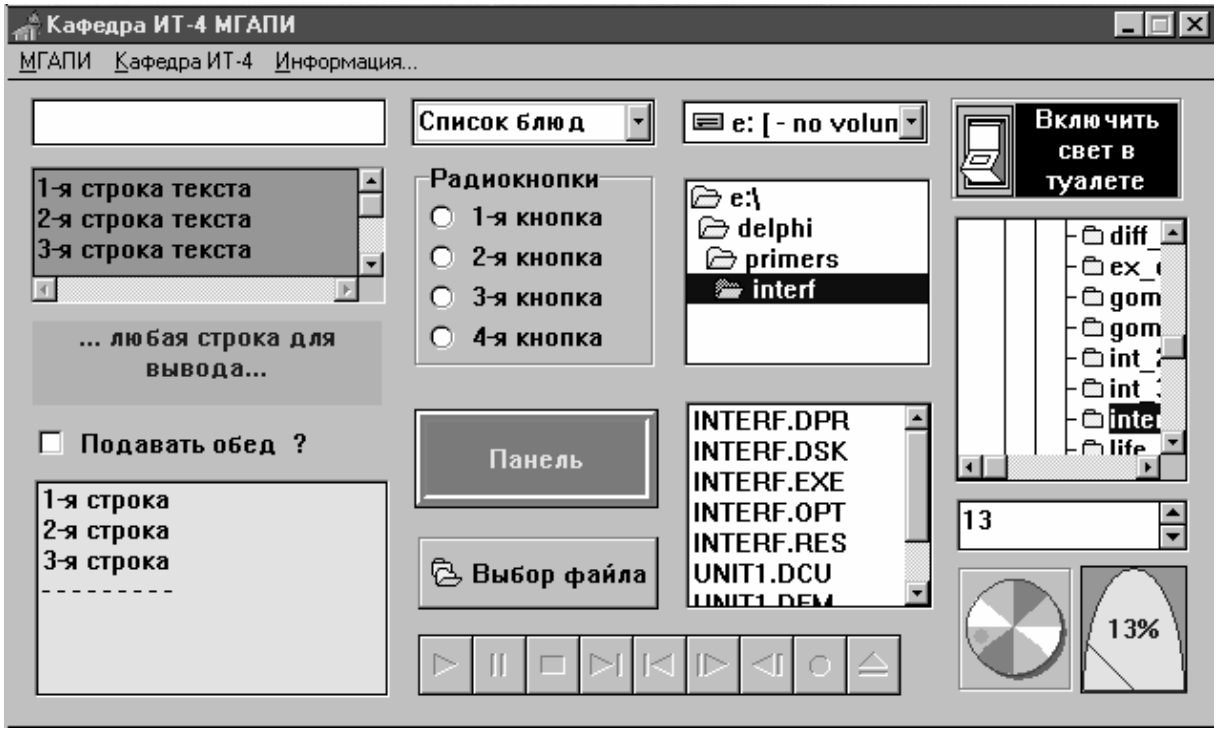


Рис.6. Форма с размещенными на ней компонентами **Delphi**

Удобно использовать правую кнопку 'мыши' для выравнивания компонентов (вариант **Scale** во всплывающем меню, предварительно следует вы-

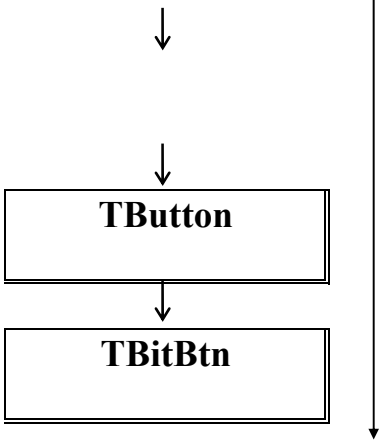
брать несколько компонентов для выравнивания их положения, включая компоненты в группу путем отметки их 'мышью' при нажатой клавише **Shift**), изменения их размеров (вариант **Size**) и масштабирования размеров (вариант **Scale**).

Большинство компонентов (визуализируемые) имеют свойства **Left**, **Top**, **Height**, **Width** (абсцисса и ордината верхнего левого угла компонента относительно родительского компонента и высота и ширина компонента в пикселах соответственно), **Hint** и **ShowHint** (текст ярлычка помощи и разрешение на его демонстрацию), **Name** (имя компонента), **Caption** (заголовок компонента), **Hide** (скрывает компонент), булево свойство **Visible** (видимость компонента), **Tag** (любое **Longint**-число для идентификации компонента), **TabOrder** (определяющее последовательность передачи фокуса - с помощью клавиши **Tab** - в пределах формы числовое значение, компонент может получить фокус только при **TabStop=TRUE**), **Handle** (получить используемый функциями WINDOWS API идентификатор данного компонента) и другие; события **OnClick**, **OnDbClick**, **OnMouseMove**, методы **ScaleBy** (масштабирует размер компонента), **Refresh**, **Repaint**, **Show** и **Update**, что является следствие наследования, свойственным объектно-ориентированной структуре объектов **Delphi** / **C++Builder**.

Для интересующихся приведём схему наследования методов для объекта **TBitBtn** (кнопка с надписью и пиктограммой).

Наиболее абстрактные (общие) методы

TObject	Наиболее общие для всех объектов методы
TPresistant	Добавлена возможность записывать себя самого в EXE-файл (при компиляции) и переносить себя обратно из EXE-файла во время выполнения
TControl	Добавлены возможности взаимодействия с пользователем
TWinControl	Добавлены возможности использования механизма WINDOWS для создания окна
TButtonControl	Обеспечивает возможность функционирования кнопок



Обеспечивает реальные кнопки возможностями нажатия, изменения надписей и др.

Обеспечивает отрисовку на кнопке пиктограммы и др.

Наиболее специфичные методы

5.1. ПОЛЕЗНЫЕ НЕВИЗУАЛЬНЫЕ ОБЪЕКТЫ Delphi / C++Builder

При работе в **Delphi** / **C++Builder** большую помощь оказывают полезные классы, служащие для упорядоченного хранения и доступа к данным (контейнер для хранения коллекции текстовых строк **TStringList**), а также часто применяемый объект выдержки времени (**TTimer**). Данные объекты являются невизуальными в том смысле, что не видны на экране дисплея во время **RunTime** (но могут быть представлены пиктограммами в **DesignTime**).

Желающих более подробно ознакомиться со стандартными компонентами **Delphi** / **C++Builder** отсылаем к книгам [8,11].

5.1.1. КЛАСС TStringList

Класс (объект) **TStringList** служит для хранения и манипуляций с набором (коллекцией) текстовых строк и является потомком абстрактного класса **TSrings**. Нижеприведенный пример объявляет объект **ListOfFamily** типа **TStringList**

```
ListOfFamily: TStringList;
```

Динамическое создание объекта производится следующим образом

```
ListOfFamily:=TStringList.Create; { создать  список  ListOfFamily }
```

Очистка списка содержащихся в объекте строк производится с помощью метода **Clear**

```
ListOfFamily .Clear;
```

Добавление строк к списку выполняет метод **Add**

```
ListOfFamily.Add('Иванов');
ListOfFamily.Add('Петров');
ListOfFamily.Add('Сидоров');
ListOfFamily.Add('Рабинович');
```

C++Builder. Соответственно

```
TStringList *ListOfFamily=new TStringList(0);  // создать объект -
                                                // список ListOfFamily
ListOfFamily->Clear();  // очистить список
ListOfFamily->Add("Иванов"); // добавить в список...
LlstOfFamily->Add("Петров");
LlstOTFamily•>Add("Сидоров");
LlstOfFamily->Add("Рабинович");
delete ListOfFamily;  // уничтожить объект ListOfFamily
```

TStringList может содержать не только строки, но и более сложные сущности - например, метод **AddObject(S:string, O:TObject)** добавляет строку **S** в паре с объектом **O**.

Строку (и комбинацию 'строка+объект') можно добавить в произвольное место в списке, для этого служат методы **Insert(Index:integer, S: string)** и **Insert0bject(lndex:integer, S:string, 0:TObject)**, вставляющие соответствующие сущности под индексом **Index**.

Метод **AddStrings(Strings:TStrings)** добавляет в конец данного набора другой набор **Strings**, метод **IndexOf(S:string)** возвращает номер в наборе строки **S** (если **S** не найдена, возвращается -1); соответствующие методы есть и для работы с объектами. Функция **Equals(Strings:TString)** сравнивает текущий список со списком **Strings**, возвращая **TRUE** в случае полного тождества объектов.

Свойство **Count** содержит число строк в объекте, к строкам и объектам имеется доступ через свойства **Strings[Index]** и **Objects[Item]**, где **Item** - номер строки (объекта), **Item** изменяется от **0** до **Count-1**.

Метод **Free** разрушает объект типа **TStringList**

```
ListOfFamily.Free;  { ... это  в  Delphi  }
```

Для загрузки/сохранения из/в дисковом файле содержимого **TStrings-List** служат процедуры **LoadFromFile(FileName:string)** и **SaveTo-File(FileName:string)**; для более подробного ознакомления рекомендуется обратиться к книге [8].

5.1.2. КЛАСС TTimer

Компонент **TTimer** (страница **System** палитры компонентов) служит для отсчета времени и уведомления программы о истечении заданного временного интервала.

Таймер начинает генерировать события **OnTimer** через интервалы време-

ни (заданного свойством **Interval** в миллисекундах) после установки булева свойства **Enabled** в **TRUE** ('остановить' таймер можно также присвоением **Value=0**).

Задержать выполнение программы на **mSecs** миллисекунд можно также с помощью следующей процедуры:

```
procedure TForm1.Delay(mSecs:  longint);
{ ждет  mSecs  миллисекунд
  ... это всё из кладезей народной мудрости
  в области WINDOWS-программирования...  }
var
   FirstTick: longint;
begin
FirstTick := GetTickCount;  {  запомнить начало отсчета  }
repeat
   Application.ProcessMessages;  { ... дать поработать другим
                                       приложениям WINDOWS !  }
until
   GetTickCount - FirstTick  >=  mSecs;   {  интервал истек ?  }
end;
```

Настоятельно обращаю внимание на обязательность применения метода **ProcessMessages** - в противном случае другие приложения WINDOWS 'повиснут' вследствие 'захвата' всех ресурсов системы данным приложением (в WINDOWS'NT не столь критично).

5.2. КОМПОНЕНТ TEdit

Компонент **TEdit** (верхний в первой слева колонке на рис.6, в палитре компонентов находится на странице **Standart**) является однострочным текстовым редактором и служит для ввода пользователем произвольной строки (которая в дальнейшем может быть преобразована, например, в число); для вывода данных используется редко.

Введенный текст содержится в свойстве **Text**. Считывание информации из компонента **TEdit** для последующего использования производится согласно следующей схемы (считая, что имя **TEdit**-компонента есть **Edit1**)

```
var  WorkString: string; { переменная типа Pascal-строки }
. . . . . .
WorkString:=Edit1.Text;
```

Часть текста в окне редактирования может быть выделена - свойства **SelStart** и **SelLength** определяют начало и длину выделенной части (в количестве символов), сам выделенный текст содержится в строке **SelText**. Метод **ClearSelection** исключает из текста выделенный фрагмент, а метод **SelectAll**

выделяет весь текст в редакторе. Свойство **Font** позволяет задать имя шрифта для символов.

Весь текст в редакторе очищается методом **Clear**, булево свойство **Modified** информирует, изменялся ли текст в процессе редактирования.

Функции **CopyToClipboard**, **CutToClipboard** и **PasteFromClipboard** позволяют работать с системным буфером обмена (при выделении части текста работа происходит именно с ней, в противном случае в операции участвует весь текст). Свойство **CharCase** позволяет задавать преобразование вводимого текста к верхнему/нижнему регистрам, **OEMConvert** - осуществлять преобразование между **OEM**- и **ANSI**-кодировками, **PasswordChar** задает символ, используемый вместо введенных при вводе пароля.

Из событий компонента **TEdit** наиболее интересны **OnChange** (генерируется при любом изменении текста), **OnEnter** (возникает при нажатии клавиши **Enter**, удобно использовать для вызова процедуры верификации ввода) и **OnClick** (возбуждается при щелчке 'мышью' на компоненте).

5.3. **КОМПОНЕНТ TMemo**

Компонент **TMemo** (второй сверху компонент в крайнем левом столбе на рис.6, в палитре компонентов находится на странице **Standart**) представляет собой многострочный текстовый редактор и служит обычно для вывода массива строк (например, столбец цифр) и включает многие свойства и сообщения компонента **TEdit**.

Содержимое объекта может быть представлено в виде набора строк **Lines** (**Lines** суть объект типа **TStrings**, содержащий текст в виде набора строк). Текст может выравниваться по левому, правому краям или по центру - свойство **Alignment**. При значении свойства **WordWrap=TRUE** при достижении вводимым текстом конца строки происходит переход на новую строку, в противном случае происходит горизонтальная прокрутка. Во время **DesignTime** можно заполнить компонент нужными строками - для этого следует использовать свойство **Lines** (ввод осуществляется в специальном открывающемся окне).

Добавление строк в **TMemo** возможно путем использования компонентной функции **Add** (работа с компонентом **Memo1**)

```
Memo1 .Lines.Add('Первый элемент');
Memo1. Lines.Add('Второй элемент');
Memo1 .Lines.Add('Третий элемент');
. . . . . .
Memo1 .Lines.Add(Edit1.Text);  {  добавить строку из Edit1  }
Memo1. Lines.Add(Edit2.Text);  {  - . - . - . - . - . - . - . - . - Edit2 }
```

Предварительно следует очистить **Memo1** путем использования метода

Clear

```
Memo1. Clear;
```

Общее число строк, содержащихся в компоненте **TMemo**, доступно через свойство **Count** (счет начинается с 0)

```
var
  NumberOfLines: Integer;
. . . . . .
NumberOf Lines: =Memo1.Lines.Count;
```

Имеется возможность как сохранить, так и восстановить содержимое **Memo1** в/из дискового файла

```
Memo1 .LInes.SaveToFile('c:\my_file.txt');
Memo1.Lines.LoadFromFile('c:\my_file.txt');
```

C++Builder. Соответственно

```
Memo1->Lines->SaveToFile("c:\\my_file.txt");
Memo1->Lines->LoadFromFile("c:\\my_file.txt");
```

Генерируемые компонентом **TMemo** события практически тождественны таковым для **TEdit**.

5.4. КОМПОНЕНТ TLabel

Компонент **TLabel** (третий сверху в левом столбце рис.6, располагается на странице **Standard** палитры компонентов) представляет собой статический текст и служит для отображения информации (часто используется при отладке).

Сам текст содержится в свойстве **Caption** (**Pascal**-строка длиной до 255 символов), пример приведен ниже (предполагается имя компонента **Label1**)

```
Label1.Caption:='Вывод любой строки';
```

C++Builder. Соответственно

```
Label1->Caption="Вывод любой строки";
```

Свойство **Alignment** устанавливает правило выравнивания текста - по правому, левому краю или по центру клиентской области, булево свойство **Autosize=TRUE** автоматически 'растягивает' размеры компонента в соответ-

ствие с размером текста и высотой шрифта, **Wordwrap** определяет возможность разрыва слов в случае превышения длиной выводимого текста ширину компонента.

Генерируемые компонентом события не вносят ничего нового по сравнению с вышеописанными.

5.5. **КОМПОНЕНТ TCheckBox**

Компонент **TCheckBox** (4-й сверху в первом столбце на рис.6, расположен на странице **Standard** палитры компонентов) является кнопкой с независимой фиксацией (флажком) и служит в качестве двоичного переключателя режимов в программе (переключается одинарным щелчком 'мыши' на компоненте).

Состояние кнопки отражается в булевом свойстве **Checked**, причем **Checked=TRUE** соответствует нажатой кнопке и наоборот. Нижеследующий пример демонстрирует проверку нажатия кнопки и соответствующее разветвление программы

```
if CheckBox1. Checked then
begin
   ... делать что-то при нажатой кнопке ...
end
else
begin
   ... что-то сделать при ненажатой кнопке ...
end;
```

Данной свойство доступно и по записи - кнопку можно 'нажать' программным путем, используя оператор

```
CheckBox1.Checked:=TRUE;
```

При установке свойства **AllowGrayed** в **FALSE** (умолчание) кнопка имеет два состояния и при каждом нажатии переходит из состояния 'нажато' в 'отжато' и обратно, при **AllowGrayed=TRUE** кнопка имеет три состояния и при нажатиях движется по циклу - 'нажато/отжато/неактивно' (соответствующие значения свойства **State** суть **cbChecked**, **cbUnchecked** и **cbCrayed** и также доступны для чтения).

Компонент генерирует события **OnClick**, **OnEnter**, **OnExit** и другие; но, к сожалению, список возможных событий не включает **OnChange** (пользователь должен анализировать состояние кнопки в обрабатывающей сообщение **OnClick** процедуре).

Кроме **TCheckBox**, имеется компонент **TRadioButton** (также располо-

женный на странице **Standard** палитры компонентов), представляющий собой кнопку с независимой фиксацией (радиокнопку); форма **TCheckButton** - круглая кнопка. Эти кнопки объединяются в группу **TRadioGroup** (см. ниже), причем только одна радиокнопка в группе может быть нажата в данный момент (при нажатии любой кнопки все другие в группе выключаются).

5.6. **КОМПОНЕНТ TListBox**

Компонент **TListBox** (нижний в крайнем левом столбце на рис.6, расположен на странице **Standard** палитры компонентов) и является списком с возможностью выбора.

Пользователь может выбрать одну из строк, хранящихся в свойстве **Items** (**Items** есть объект типа **TStrings**, содержащий текст в виде набора строк), индекс текущего (сфокусированного) элемента списка содержится в свойстве **Itemlndex**. При установке свойства **MultiSelect** в **FALSE** в списке не может быть выделено несколько элементов одновременно и значение свойства **ExtendedSelect** не играет роли. При **MultiSelect=TRUE** (может быть выделено несколько элементов одновременно) в случае **ExtendedSelect=FALSE** при каждом щелчке 'мыши' изменяется выделение только сфокусированного элемента, в случае **ExtendedSelect=TRUE** выбор происходит при передвижении 'мыши' с нажатой левой кнопкой на новом элементе списка при нажатых клавишах **Ctrl** или **Shift** или при нажатии **Shift+**стрелки.

Количество выделенных элементов содержится в свойстве **SelCount**, проверить и установить выделение для элемента с номером Index можно путем обращения к булеву свойству **Selected[Index]**.

Сортировка строк в алфавитном порядке достигается установкой свойства **Sorted** в **TRUE**; к сожалению, не имеется свойства обеспечения уникальности строк.

Элементы списка могут располагаться не только в одном столбце, но и в нескольких - число столбцов задается свойством **Columns**. Подгонка высоты данного компонента с целью помещения целого числа элементов достигается установкой **IntegralHeight=TRUE**.

Вывод в **ListBox1** квадратных корней первых 100 чисел может быть осуществлен следующей последовательностью операторов (при вычислении квадратного корня неявно производится преобразование 'integer□ float')

```
var
   I: integer;
   Int,Float: string;
. . . . . .
ListBox1 .Clear;  { очистить список }
for I:=1 to 100 do
begin
```

```
   Str(I:4, Int);  { форматирование целого числа - 4 позиции на число}
   Str(Sqrt(I):15:5, Float);  { форматирование вещественного числа -
                                всего 15 позиций, из них 5 -
                                для дробной части числа }
   ListBox1.Items.Add(Int + ':' + Float);
end;
```

Число строк в **ListBox** можно получить, прочитав свойство **Items.Count**; ввести строки в **ListBox** во время **DesignTime** можно, щелкнув 'мышью' по кнопке справа от строки **Items** в Инспекторе Объектов.

Генерируемые компонентом **TListBox** события включают **OnClick**, **OnDbClick** и другие; для разработчика представляет интерес событие **OnDrawItem**, возникающее при перерисовке каждого элемента (обычно процедура обработки этого события дополняет пиктограммами строки списка).

5.7. **КОМПОНЕНТ TComboBox**

Компонент **TComboBox** (верхний в центральном столбце на рис.6, расположен на странице **Standard** палитры компонентов) и является выпадающим списком с возможностью выбора и редактирования. Фактически **TComboBox** представляет собой выпадающий при нажатии 'мышью' на кнопку со стрелкой вниз собственно список (подобно **TListBox**), дополненный полем ввода (подобно **TEdit**). С помощью компонента **TComboBox** пользователь может выбрать одно из имеющихся значений списка или ввести свое. Заменяя **TListBox**, компонент **TComboBox** имеет больше возможностей и требует значительно меньше пространства на форме для размещения.

Как и для **TListBox**, доступ к элементам списка достигается через свойство **Items** (**Items** есть объект типа **TStrings**, содержащий текст в виде набора строк), далее см. все описания для **TListBox**.

Содержащийся в редактируемом элементе текст доступен через свойство **Text**, также доступны свойства **SelText**, **SelStart**, **SelLength** и **SelectAll** (см. описание компонента **TEdit**).

Значение булева свойства **DroppedDown** (только для **RunTime**) соответствует состоянию списка, максимальное число показывающихся при выпадении списка элементов задается свойством **DropDownCount** (по умолчанию 8).

При изменении текста в окне редактирования возникает событие **OnChange**, при изменении состояния списка (выпавший/скрытый) возникает событие **OnDropDown**; также генерируются события **OnClick** и **OnDbClick** и др.

5.8. **КОМПОНЕНТ TRadioGroup**

Компонент **TRadioGroup** представляет собой группу радиокнопок с зависимой фиксацией (компонентов **TRadioButton**), в палитре компонентов находится на странице **Standard**.

Во время проектирования (**DesignTime**) с помощью Инспектора Объектов в свойстве **Items** следует задать текст описания для каждой из кнопок (на рис.6 тексты суть **RadioButton1**, **RadioButton2÷RadioButton4**), число строк текста и определит число радиокнопок.

Кнопки могут располагаться в несколько столбцов (свойство **Columns**), индекс нажатой кнопки определяется свойством **ItemIndex** (начиная с 0, при **Itemlndex=-l** ни одна из кнопок не нажата), набор строк с заголовками радиокнопок содержится в свойстве **Items** (объекты типа **TSrings**). Список сообщений компонента **TRadioGroup** довольно беден и не приводится; интерес представляет собой сообщение **OnClick** (щелчок 'мышью' в пределах компонента).

Пример связанной с событием **OnClick** процедуры обработки состояния второй из показанных на рис.6 кнопки приведен ниже

```
procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
  if RadioGroup1.ItemIndex = 1 then { вторая в группе кнопка включена )
  begin
    MessageBeep(MB_OK); { подать звук... }
    RadioGroup1.Items[1]:='Ой ! Меня нажали !'; { надпись на кнопке }
    ... сделать что-то, зависящее от нажатия второй радиокнопки ...
    end;
end; { Баканов В.М., МГАПИ, кафедра ИТ-4, 1996-2000 }
```

5.9. КОМПОНЕНТ TPanel

Компонент **TPanel** является несущей конструкцией для размещения на ней других элементов управления, являясь в этом случае родителем для размещенных на ней компонентов. Настоятельно рекомендуется использовать компонент **TPanel** для размещения компонентов при создании пользовательского интерфейса !

Внешнее оформление панели определяется свойствами **BevelInner**, **BevelOuter** (возможные значения **bvNone**, **bvLovered** и **bvRaised**), **BevelWidth**, **BorderWidth**, **BevelInner** и **BevelOuter** представляют собой (внутреннюю и внешнюю) окаймляющие панель рамки, имитирующие 'приподнятость' (**bvRaised**) или 'утопленность' (**bvLovered**) шириной **BevelWidth**. Иллюзия трехмерности также создается изменением свойства **BorderStyle**.

Свойство **Alignment** определяет горизонтальное выравнивание содержащегося в свойстве **Caption** текста заголовка панели.

Список генерируемых событий включает **OnClick**, **OnDbClick** и др., событие **OnResize** возникает при изменении размеров панели.

5.10. **КОМПОНЕНТ TBitBtn**

Компонент **TBitBtn** представляет собой кнопку с пиктограммой и текстом (на рис.6 данный компонент является 4-м сверху в среднем ряду, в палитре компонентов расположен на странице **Additional**). В отличие от родственных компонентов **TButton** и **SpeedButton** данный компонент имеет ряд дополнительных удобств в использовании.

В **Delphi** определены стандартные типы кнопок, определяемые свойством **Kind**; для каждой из них определены передаваемый форме результат, название и картинки. Достаточно установить нужное значение **Kind**, и кнопка приобретет нужный вид. Например, в случае **Kind=bkHelp** нажатие кнопки инициализирует систему помощи, **bkClose**-кнопка закрывает форму.

При **Kind=bkCustom** параметры кнопки определяются пользователем. Стиль кнопок задается свойством **Style**, текст на кнопке - свойством **Caption**, располагаемая на теле кнопки пиктограмма - **Glyph**, промежуток между пиктограммой и текстом - **Spacing** и т.д. Присвоение свойству **Enabled** значения **FALSE** (в **RunTime** также) дезактивирует кнопку (устанавливает ее в неактивное 'серое' состояния); эта возможность удобна, например, при необходимости деактивировать некоторые кнопки, не задействованные в данном режиме работы программы.

Из событий наиболее часто используется **OnClick**, также генерируются **OnKeyDown**, **OnKeyPress**, **OnKeyUp** и др.

Из компонентных методов представляют интерес **Click** (программная имитация нажатия кнопки).

Процедура, например, 'отключающая' кнопку **BitBtn1** после нажатия кнопки **BitBtn2**, может выглядеть следующим образом

```
procedure TForm1.BitBtn2Click(Sender: TObject);
begin
with BitBtn1 do
  begin
    Caption:='Прощай навсегда ...'; { текст на кнопке ...}
    Enabled:=FALSE; { деактивировать кнопку }
  end;
end; { Баканов В.М., МГАПИ, кафедра ИТ-4,1996-2000 }
```

Вновь активировать эту кнопку можно только присваиванием **BitBtn1.Enabled:=TRUE.**

Специальная кнопка (компонент **TSpeedButton**) может иметь как зависи-

мую, так и независимую фиксацию (при значении свойства **GroupIndex=0** кнопка не имеет фиксации в нажатом состоянии и она не зависит от остальных кнопок, кнопки же с одинаковым ненулевым значением свойства **GroupIndex** имеют зависимую в пределах одного родительского элемента фиксацию).

Поведение этих кнопок зависит также от булева свойства **AllowAllUp** (при **AllowAllUp=FALSE** нажатую кнопку в группе можно отпустить лишь путем нажатия другой, в противном случае кнопку можно отпустить путем повторного нажатия). Для возможности фиксации выбранной кнопки ей необходимо присвоить уникальный групповой индекс (в свойстве **GroupIndex**) и установить **AllowAllUp=TRUE**.

Компонент **TSpinButton** представляет собой две кнопки со стрелками вверх/вниз и служит для управления некоей внешней величиной (путем обработки событий **OnUpClick** и **OnDownClick** - нажатие кнопок вверх/вниз соответственно).

Программа имеет возможность проверить нажатие кнопки, прочитав свойство **Down**.

5.11. КОМПОНЕНТ TMediaPlayer

Компонент **TMediaPlayer** служит для управления мультимедийными устройствами (расположен в нижней части центрального столбца на рис.6, в палитре компонентов находится на странице **System**) и представляет с точки зрения пользователя набор кнопок для (интерактивного) управления устройством путем посылки команд **MCI -** *Media Control Interface* [3]. Программист имеет возможность модифицировать реакцию на нажатия кнопок (часть или все кнопки могут быть сделана невидимыми) и другие события в зависимости от специфики задачи.

Тип мультимедийного устройства определяется свойством **DeviceType** (для автоматического распознавания типа следует выбрать **dtAutoSelect**), набор возможностей устройства определяется свойством **Capabilities**.

Булево свойство **VisibleButtons** управляет видимостью отдельных кнопок линейки управления мультимедийным устройством, свойство **Display** указывает на окно, в котором будет происходить отображение видеоинформации (в случае **Display=NIL** создается собственное окно), **DisplayRect** задает прямоугольную область экрана для изображения.

Управление мультимедиа-устройством не обязательно должно происходить путем нажатия кнопок на линейке управления, линейка может быть скрыта (свойство **Visible=FALSE**), а управление осуществляться программой с использованием компонентных функций **Start**, **Stop**, **Pause**, **Rewind** и др. (названия методов совпадают с именами соответствующих кнопок управляющей линейки).

Вследствие большого количества свойств и событий данного компонента отсылаем заинтересованных к книгам [5,8].

Нижеследующая процедура позволяет выбрать нужный видеофайл из заданных маской в свойстве **Filter** компонента **OpenDialog1** (см. ниже) и проиграть его, сообщая об ошибке в случае возникновения оной

```
procedure TForm1.BitBtn1Cllck(Sender: TObject);
begin
  OpenDialog1.Filter := 'Все файлы|*.*|AVI/1-файлы|*.avi|' +
                        'WAV-файлы|*.wav|MIDI-файлы|*.mid';
  with MediaPlayer1 do
  begin
    try
    if OpenDialog1.Execute then
      FileName:=OpenDialog1.Filename;
    Open;
    except
      MessageDlg(MediaPlayer1.ErrorMessage,
                 mtError, [mbOk], 0);
    end;
  end;
end;
```

5.12. КОМПОНЕНГ TDriveComboBox

Компонент **TDriveComboBox** (верхний в третьем слева столбце на рис.6) является специализированным комбинированным списком (**TComboBox**) и служит для информации об имеющихся в системе устройствах прямого доступа и выбора необходимого. Список устройств заносится в компонент при инициализации оного, во время выполнения программы отображаются имя устройства, метка устройства (в квадратных скобках) и пиктограмма типа устройства (винчестер, сетевой диск, CD-ROM дисковод etc). Обычно данный компонент применяется в сочетании с нижеописанными **TDirectoryListBox** и **TFileListBox** и имеет специальные средствами связи с ними.

Имя текущего (выбранного пользователем) диска соответствует значению свойства **Drive**, а имя с меткой (как и показывается в списке) - в свойстве **Text**.

Обработчик зн**а**чимого события **OnChange** может выглядеть так

```
var CurrentDrive: string;
. . . . . .
procedure TForm1.0nDriveChange(Sender: TObject);
begin
   CurrentDrive:=DriveComboBox1.Drive; { запомнить в переменной
                                         CurrentDrive имя диска }
end;
```

5.13. **КОМПОНЕНТ TDirectoryListBox**

Компонент **TDirectoryListBox** (второй сверху в третьем слева столбце на рис.6) является специализированным списком (**TListBox**) и служит дня показа и выбора списка каталогов на текущем устройстве (устройство задается свойством **Drive**, каталог на нем - свойством **Directory**). Свойство **DirLabel** может быть установлено на реальное имя компонента **TLabel** (для визуализации выбранного каталога).

Полный путь к каталогу можно получить при помощи метода **GetItemPath(Index)**, где **Index** - номер каталога в списке (начиная с 0).

При наличии на форме компонента **TDriveComboBox** можно связать его с **TDirectoryListBox** путем установки реального имени **TDirectoryListBox** в свойстве **DirList** компонента **TDriveComboBox**.

Зн**а**чимое событие - **OnChange**.

5.14. **КОМПОНЕНТ TFileListBox**

Компонент **TFileListBox** (третий сверху в третьем слева столбце на рис.6) является специализированным списком (**TListBox**) и служит для показа и выбора файлов текущего каталога текущего устройства (устройство задается свойством **Drive**, каталог на нем - свойством **Directory**).

Свойство **FileName** содержит имя текущего файла, в свойстве **Mask** задается шаблон выбора файлов, фильтр файлов по атрибутам их реализуется соответствующей установкой свойства **FileType**. Свойство **FileEdit** может быть установлено на реальное имя компонента **TEdit** (для индикации выбранного файла).

Метод **ApplyFilePath(Path:string)** переустанавливает путь согласно строке **Path** (также переустанавливаются связанные **TDriveComboBox** и **TDirectoryListBox**).

Связать данный **TFileListBox** с установленным на форме **TDirectoryListBox** можно, установив свойство **FileList** компонента **TDirectoryListBox** на реальное имя компонента **TFileListBox**.

Зн**а**чимое событие - **OnChange**.

5.15. **КОМПОНЕНТ BiSwitch**

Компонент **BiSwitch** является **VBX**-компонентом (необходим д**о**ступ к файлу **VBX.DLL**), выполняет функцию двоичного переключателя (нажатие 'мышью') и расположен на странице **VBX** палитры компонентов (на рис.6 - верхний в правой колонке).

Из свойств отметим **TextPosition** (задает расположение текста) и **pOn**

(при **pOn=TRUE** исходное состояние переключателя включенное - отличается красной меткой, в противном случае - наоборот).

Представляющие особый интерес события **OnOff** и **OnOn** возникают в случае выключения и включения переключателя соответственно.

VBX (*Visual Basic eXtensions*) -технология (основанная на **Visual Basic** - компонентах) объявлена Microsoft Corp. пройденным этапом в развитии компонентной архитектуры WINDOWS и не поддерживается в **Delphi 2.0** и выше; вместо них используются элементы **OCX/ActiveX**. Таким образом, находящиеся на странице **VBX** ИС **Delphi 1.0** компоненты **BiSwitch**, **BiGauge**, **BiPict** и **ChartFX** нельзя использовать при переходе на **Delphi 2.0** / **C++Builder**.

5.16. **КОМПОНЕНТ TSpinEdit**

Компонент **TSpinEdit** (второй сверху в правом крайнем столбце, страница **Samples** палитры компонентов) является удачной комбинацией **TEdit** и **TSpinButton** и служит для редактирования (возможно, даже без ввода чисел с клавиатуры) целой величины (свойство **Value** доступно по чтению и записи) путем щелка 'мышью' по кнопкам со стрелками вверх/вниз.

Свойство **Increment** задает шаг изменения, а **MinValue** и **MaxValue** задают диапазон изменения управляемой величины, значение свойства **ReadOnly=TRUE** запрещает редактирование управляемой величины с клавиатуры.

Зн**а**чимые события - **OnClick** и **OnChange**.

5.17. **КОМПОНЕНТ TDirectoryOutline**

Компонент **TDirectoryOutline** (третий сверху в крайнем справа столбце на рис.6, страница **Samples** палитры компонентов) служит для отображения дерева файлов и каталогов текущего диска (заменяя **TDirectoryListBox** и **TFileListBox**).

Значения текущего диска задаются в свойстве **Drive**, каталога - **Directory**. При **TextCase=tcLowerCase** происходит преобразование символов к нижнему регистру, при **TextCase=tcUpperCase** - к верхнему, при **TextCase=tcAsIs** (умолчание) - преобразования не происходит.

Пользователь путем двойного щелчка 'мыши' имеет возможность разворачивать и сворачивать выбранные ветви дерева файловой структуры (при этом возникают события **OnExpand** и **OnCollapse** соответственно).

Другие зн**а**чимые события - **OnClick** и **OnChange**.

Ниже приведены две функции, обрабатывающая событие **OnClick** компонента **TDirectoryOutline** (первая) и событие **OnChange** компонента **DriveComboBox** (вторая); обе настраивают компонент **DirectoryOutline** на показ

файловой структуры выбранного диска.

```
procedure TForm1.DirectoryOutline1Click(Sender: TObject);
begin                                DirectoryOutLIne1.Drive:=DriveComboBox1.
  Drive;
end;

procedure TForm1.DriveComboBox1Change(Sender: TObject);
begin
  DirectoryOutLine1.Drive:=DriveComboBox1.Drive;
end;
```

5.18. **КОМПОНЕНТ TGauge**

Компонент **TGauge** (самый нижний в крайнем справа столбце на рис.6, в палитре компонентов находится на странице **Samples**) моделирует индикатор, отражающий значение некоей величины в процентах; обычно применяется для динамического отображения процента выполнения длительно протекающего процесса (например, копирования данных).

Стиль компонента задается свойством **Kind**, возможные значения - **gkText** (текстовый вывод величины в процентах), **gkHorizontalBar** (вертикальное заполнение), **gkVerticalBar** (вертикальное заполнение), **gkPie** (отклонение 'стрелки спидометра', см. рис.6) или **gkNeeddle** (заполнение сектора окружности).

Свойства **MinValue**, **MaxValue**, **Progress** и **PercentDone** определяют минимальное и максимальное значение шкалы измерения и текущее значение индикатора (абсолютное и относительное в процентах) соответственно, метод **AddProgress(VaIue: longint)** добавляет **Value** к **Progress**.

5.19. **КОМПОНЕНТ TImage**

Компонент **TImage** (страница **Additional** палитры компонентов) предназначен для показа на форме изображения - карты битов (**TBitmap**), метафайла (**TMetaFile**) или иконки (**TIcon**).

Свойство **Picture** служит контейнером для графического объекта одного из перечисленных классов, булево свойство **AutoSize=TRUE** настраивает компонент по размерам содержащегося в нем графического объекта, при **Stretch=TRUE** изображение заполняет всю область компонента (масштабируясь при необходимости). При **Stretch=FALSE** и **Center=TRUE** изображение центрируется в пределах рабочей области (**Center=FALSE** размещает изображение в левой верхней части оной).

Чтение и сохранение на диск файла изображения достигается использованием функций Picture-контейнера **LoadFromFile** и **SaveToFile** соответствен-

но.

Ниже приведен текст процедуры, позволяющей выбрать и продемонстрировать изображение из выбранного файла

```
procedure TForm1.BltBtn1Cllck(Sender: TObject);
begin
  OpenDialogl.Filter :='Все файлы|***|BMP-файлы|*.bmp|'+
                       WMF-файлы|*.wmf|ICO-файлы|*.ico';
  with Image1 do
  begin
  try
    if OpenDialog1.Execute then
      Picture.LoadFromFile(OpenDialog1.Filename);
  except
    MessageDlg('Ошибка  демонстрации изображения из файла ' + OpenDia-
                log1.Filename,
                mtError,  [mbOk],  0);
  end;
  end;
end; { конец процедуры }
```

5.20. СТАНДАРТИЫЕ ДИАЛОГОВЫЕ ОКНА WINDOWS И ИХ ПРИМЕНЕНИЕ

В списке компонентов **Delphi** / **C++Builder** имеются объекты, позволяющие работать со стандартными окнами диалога WINDOWS; в **DesignTime** они представляются виде соответствующих пиктограмм (иконок), служащих для выбора компонента с целью редактирования его свойств и сообщений с помощью **Object Inspector**'а.

Находятся эти компоненты на странице **Dialogs** палитры компонентов и инициализируются (в частности, визуализируются в виде диалоговых окон) функцией **Execute**, возращающей **TRUE** в случае выбора файла или **FALSE** в случае отказа от выбора (нажатия кнопки **Cancel**); таким образом настоятельно рекомендуется пользоваться конструкцией вида (вместо **XXXXX** подставляется реальное имя компонента)

```
if XXXXX.Execute then
   ... что-то делать ...
else
   exit;
```

5.20.1. КОМПОНЕНТ TOpenDialog

Компонент **TOpenDialog** служит для выбора файла с целью его последующей обработки (диалоговое окно компонента приведено на рис.7).

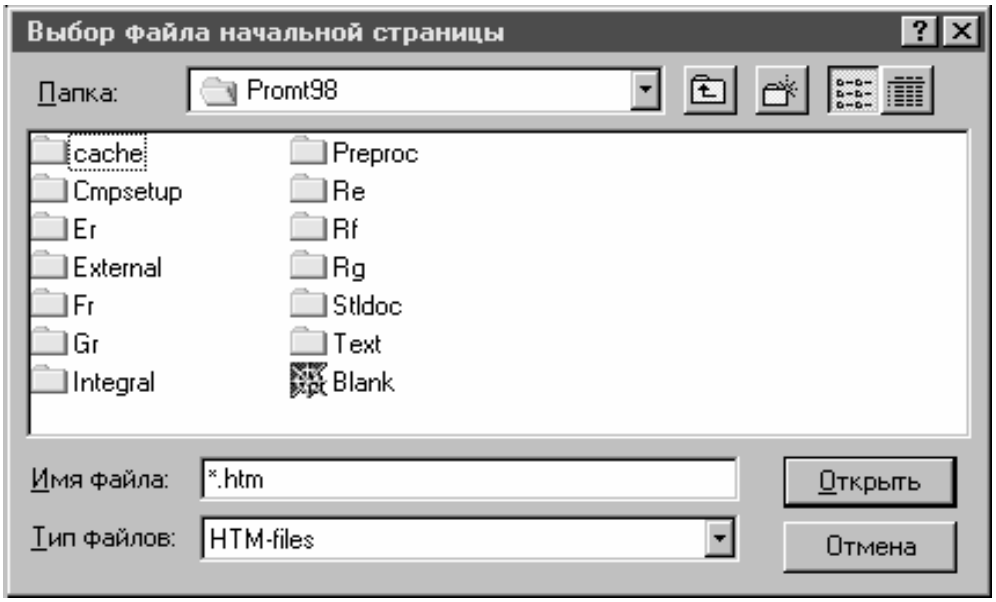


Рис.7. Диалоговое окно компонента **TOpenDialog**

В свойство **Filter** заносятся (с помощью специальной раскрывающейся панели **Object Inspector**'а) расширения файлов, соответствующие маске поиска (совместно с текстовым описанием каждой маски), свойство **FilterIndex** указывает, какая из масок будет текущей при появлении диалога на экране, в свойство **Title** заносится заголовок диалогового окна, в **InitialDir** - имя желаемого каталога (если пусто, отображается содержимое текущего каталога), свойство **DefaultExt** задает расширение файла по умолчанию.

Установка свойства **FileEditStyle** в **fsEdit** (умолчание) соответствует приведенному рисунку, вариант **FileEditStyle=fsComboBox** вызывает появление содержащего историю выбора файлов выпадающего списка.

Выбранный файл доступен в свойстве **FileName**, событий компонент не имеет.

5.20.2. **КОМПОНЕНТ TSaveDialog**

Компонент **TSaveDialog** служит для выбора имени сохраняемого файла (диалоговое окно компонента приведено на рис.8).

Свойства данного компонента повторяют таковые для **TOpenDialog**, событий нет.

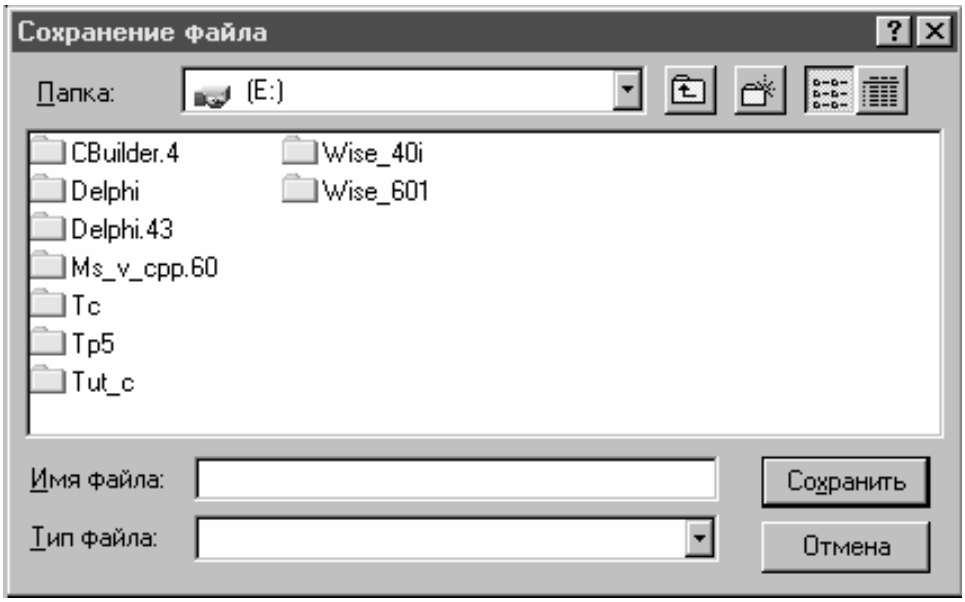


Рис.8. Диалоговое окно компонента **TSaveDialog**

5.20.3. **КОМПОНЕНТ TFontDialog**

Компонент **TFontDialog** служит для выбора шрифта (очертания символов), выбранное значение содержится в свойстве **Font** (диалоговое окно компонента приведено на рис.9).

Свойство **Device** определяет тип устройства, для которого выбираются шрифты. **Options** задает режимы диалоговой панели, **MinFontSize** и **MaxFontSize** ограничивают высоту выбираемых шрифтов. Например, в следующем примере показано, как отобразить список только **TrueType**-шрифтов и присвоить выбранный шрифт компоненту **Memo_1** (т.е. изменить шрифт отображения строк в **Memo_1**)

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  with FontDialog1 do
  begin
    Options:=[fdTrueTypeOnly];  { только TrueType фонты  ! }
    if Execute then
    Memo_1.Font:=Font;  { собственно присваивание типа шрифта }
  end;
end;
```

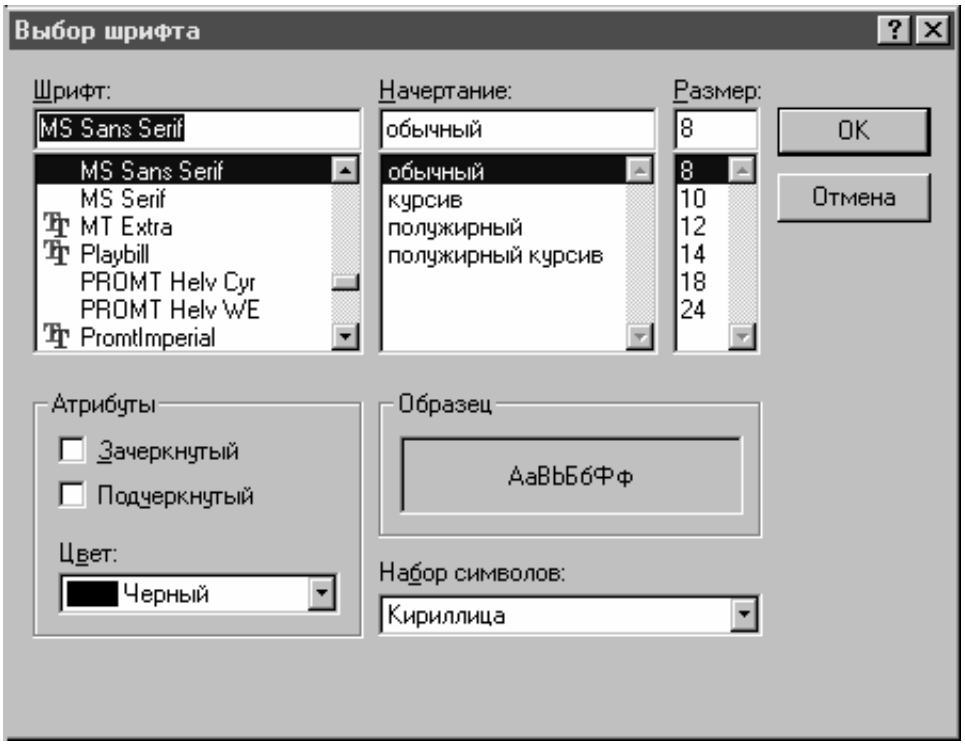


Рис.9. Диалоговое окно компонента **TFontDialog**

Присваивание типа шрифта можно проводить в обработчике события **OnApply**, возникающего при нажатии кнопки **Apply** на данной панели (кнопка **Apply** появляется на панели только при назначении обработчика данного события).

5.20.4. **КОМПОНЕНТ TColorDialog**

Компонент **TColorDialog** служит для выбора цвета, выбранное значение содержится в свойстве **Color** (диалоговое окно компонента приведено на рис.10).

Например, в нижеследующем примере показано, как ‘перекрасить’ форму **Form1** в выбранный (заданный глобальной переменной **Color**) цвет

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  with ColorDialog1 do
  begin
    If Execute then
      Form1.Color:=Color;
  end;
end;
```

Событий компонент **TColorDialog** не имеет.

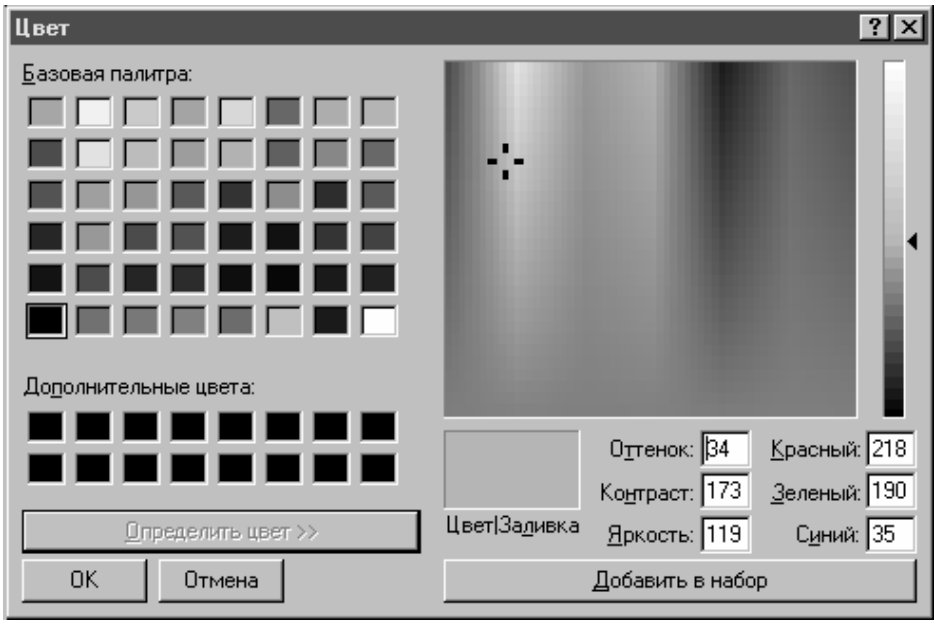


Рис.10. Диалоговое окно компонента **TColorDialog**

5.20.5. **КОМПОНЕНТ TPrintDialog**

Компонент **TPrintDialog** служит для задания характеристик принтера, событий не имеет (диалоговое окно компонента приведено на рис.11).

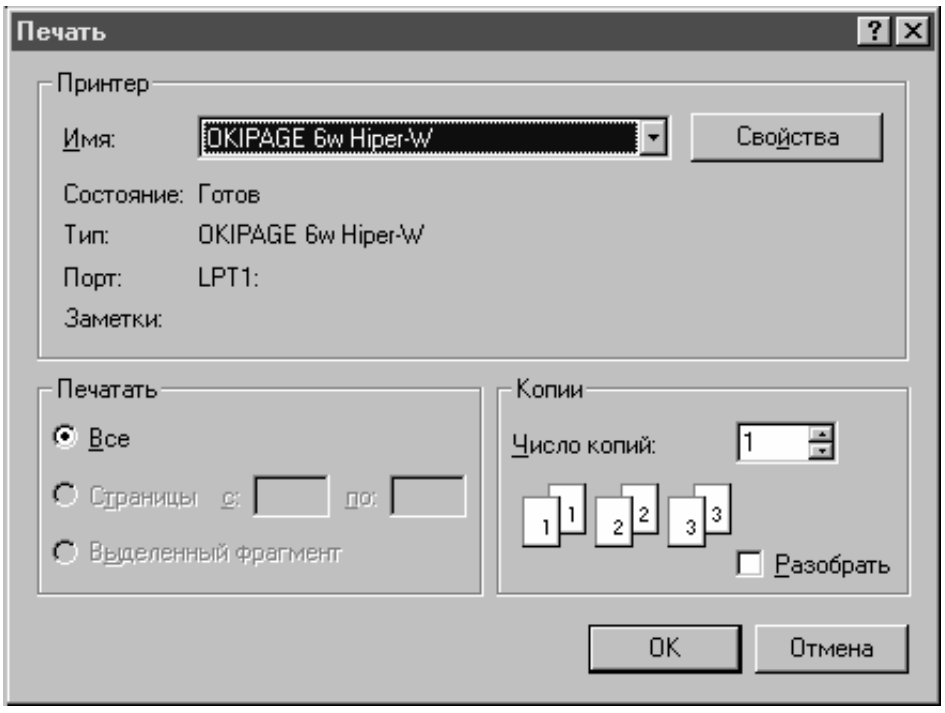


Рис.11. Диалоговое окно компонента **TPrintDialog**

5.20.6. **КОМПОНЕНТ TPrintSetupDialog**

Компонент **TPrintSetupDialog** служит для выбора текущего принтера и установки режимов его работы, событий не имеет (диалоговое окно компонента приведено на рис.12).

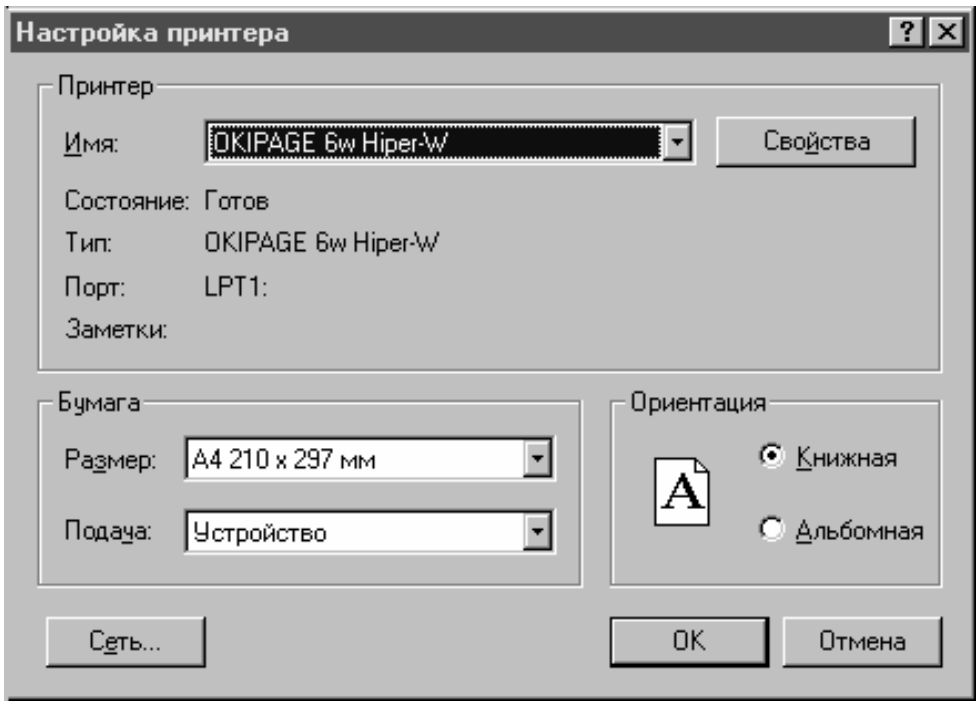


Рис.12. Диалоговое окно компонента **TPrintSetupDialog**

5.20.7. **КОМПОНЕНТ TFindDialog**

Компонент **TFindDialog** служит для вывода стандартной панели ввода образца и инициализации процесса поиска его в последовательности символов (диалоговое окно компонента приведено на рис.13).

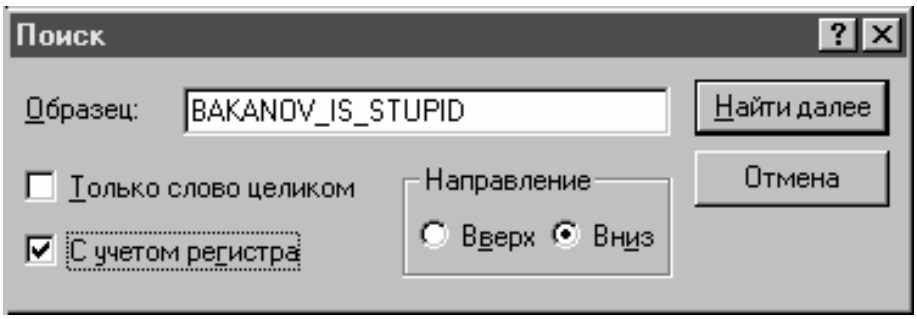


Рис.13. Диалоговое окно компонента **TFindDialog**

Искомый образец находится в свойстве **FindText**; при нажатии кнопки **FindNext** возникает событие **OnFind**, в обработчике которого и должен происходить собственно поиск по образцу.

5.20.8. **КОМПОНЕНТ TReplaceDialog**

Компонент **TReplaceDialog** служит для вывода стандартной панели ввода образцов '*заменить что...*' и '*заменить на...*', инициализации процессов поиска первого в последовательности символов и замены на второй (диалоговое окно компонента приведено на рис.14).

Искомый образец находится в свойстве **FindText**, образец для замены - в **ReplaceText**. При нажатии кнопки **FindNext** возникает событие **OnFind**, в обработчике которого должен происходить собственно поиск по образцу; при нажатии кнопок **Replace** или **ReplaceAll** возникает событие **OnReplace**, в обработчике которого должна происходить непосредственная замена.

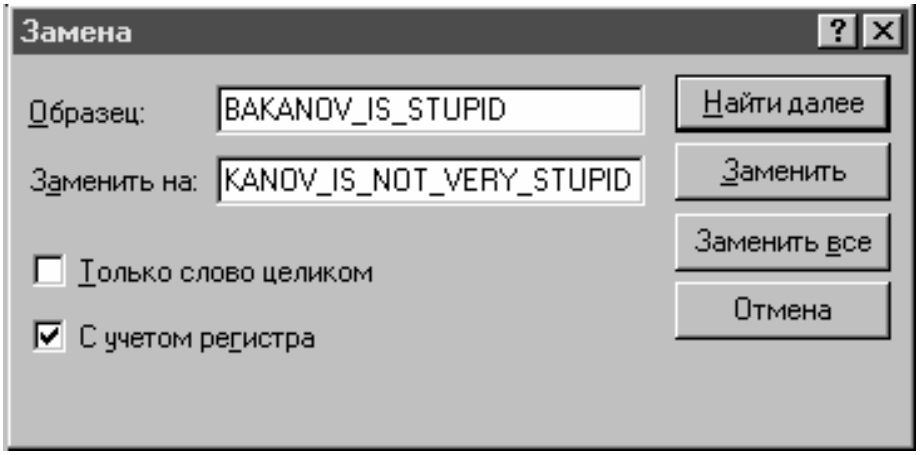


Рис.14. Диалоговое окно компонента **TReplaceDialog**

5.21. **ДОПОЛНИТЕЛЬНЫЕ КОМПОНЕНТЫ Delphi И C++Builder**

Стандартная поставка **Delphi** и **C++Builder** включает около 200 штатных компонентов. Однако б**о**льшая часть мощности указанных интегрированных ср**е**д заключается именно в возможности (практически неограниченного) наращивания функциональности ИС путем расширения числа компонентов. В настоящее время в мире (в т.ч. в сети InterNet) доступны тысячи компонент для **Delphi** и **C++Builder**, обеспечивающие практически любое возможное применение указанных ИС.

Любой (достаточно квалифицированный) пользователь **Delphi** / **C++Builder** может создать свой собственный компонент (при уверенности в нужности и востребованности последнего); для интересующихся рекомендуем книгу [12].

Ниже приведены некоторые WEB-адреса, представляющие интерес для разработчиков компонентов для **Delphi** и **C++Builder**

- http://www.intersurf.com/infiniti
- http://www.regatta.com
- http://www.wol2wol.com
- http://www.tpower.com
- http://www.eshalon.com

Delphi версий выше 1.0 и **C++Builder** предоставляют (в стандартной поставке) множество компонентов, привычных (по визуальному представлению) пользователям WINDOWS'9x и WINDOWS'NT; данные компоненты расширяют функциональность вышеописанных и расположены на странице **Win32** палитры компонентов.

Например, можно рекомендуется использование компонента **TTreeView** вместо **TOutline**, **TProgressBar** вместо **TGauge** и др. Вышеприведенный список не включает описания компонентов, необходимых для создания **ActiveX**-объектов и **InterNet**-приложений; рекомендуются работа [10] и др.

Для пользователей **Delphi** могут быть интересны издания, посвященные данной ИС

- http://www.cobb.com/ddj/index.htm
- http://www.informant.com/undu/index.htm
- http:///www.members.aol.comdelphimag
- http://www.teleport.com/~ol/djournal.shtml
- http://www.informant.com/delphi/di_index.htm

Указанные WEB-сайты не могут, естественно, полностью представить информацию о **Delphi** в сети InterNet; заинтересованным рекомендуем произвести поиск в Сети (применяя соответствующие поисковые системы) по соответствующим ключевым словам.

Существует и более современный путь расширения библиотеки визуальных компонент - например, загрузка дополнительных компонент в виде **ActiveX** с WEB-адресов службы поддержки интегрированных сред **Delphi** и **C++Builder**.

5.22. ПОДДЕРЖКА ТЕХНОЛОГИЙ DDE И OLE В Delphi И C++Builder

Поддержка стандартных для WINDOWS технологий **DDE** и **OLE** инкапсулирована в компонентах **TDDEClientConv**, **TDDEClientItem**, **TDDEServerConv**, **TDDEServerItem** и **OLEContainer** соответственно; поддержка

OLEAutomation достигается с помощью эксперта **Automation Object Expert** (функции поддержки описаны в модуле **OLE2.PAS**), подробнее см. [8,10].

6. СТАНДАРТНЫЕ МЕНЮ WINDOWS

Delphi предоставляет пользователю компоненты, реализующие служащие для выбора нужного действия или режима стандартные меню WINDOWS - **TMainMemu** (располагаемое в верхней части формы меню в виде горизонтальной линейки с выпадающими вниз вложенными пунктами меню, в одной форме может быть только одно меню такого типа) и **TPopupMenu** (всплывающее по щелчку правой кнопки 'мыши' меню в стиле WINDOWS'9; в пределах одной формы этих меню может быть несколько - индивидуально для каждого компонента - в этом случае можно говорить о контекстном **PopUp**-меню).

Оба компонента располагаются на странице **Standard** палитры компонентов и должны быть зарегистрированы (с помощью **Object Inspector**'а или во время **RunTime**) в свойствах **Menu** и **PopupMenu** формы-владельца для обоих типов меню и/или в свойстве **PopupMenu** конкретного компонента для **PopUp**-меню.

Для желающих получить более изощренные сведения о возможностях меню рекомендуется книга [8].

6.1. КОМПОНЕНТ TMainMenu

Для проектирования главного меню следует 'положить' на форму компонент **TMainMenu** и с помощью **Object Inspector**'а настроить свойства объекта. При нажатии расположенной справа в строке **Items** кнопки вызывается окно проектирования меню (см. рис.15), в котором пользователь вводит текст пунктов меню в свойство **Caption** (ввод дефиса вызывает появление горизонтальной разделительной черты между пунктами меню, знак '&' позволяет использовать следующий за ним символ для быстрого доступа к данному пункту меню); булево свойство **Checked** позволяет замаркировать данный пункт меню 'галочкой', свойство **ShortCut** - выбрать 'горячую' клавишу (или сочетание клавиш) для быстрого доступа к данному пункту меню.

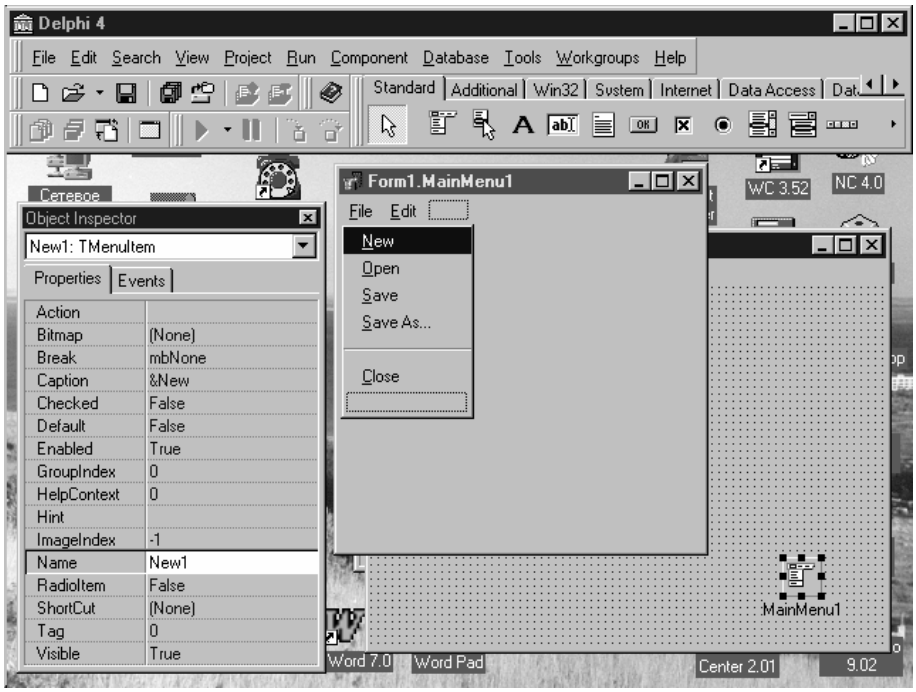


Рис.15. Проектирование главного меню (**TMainMenu**)

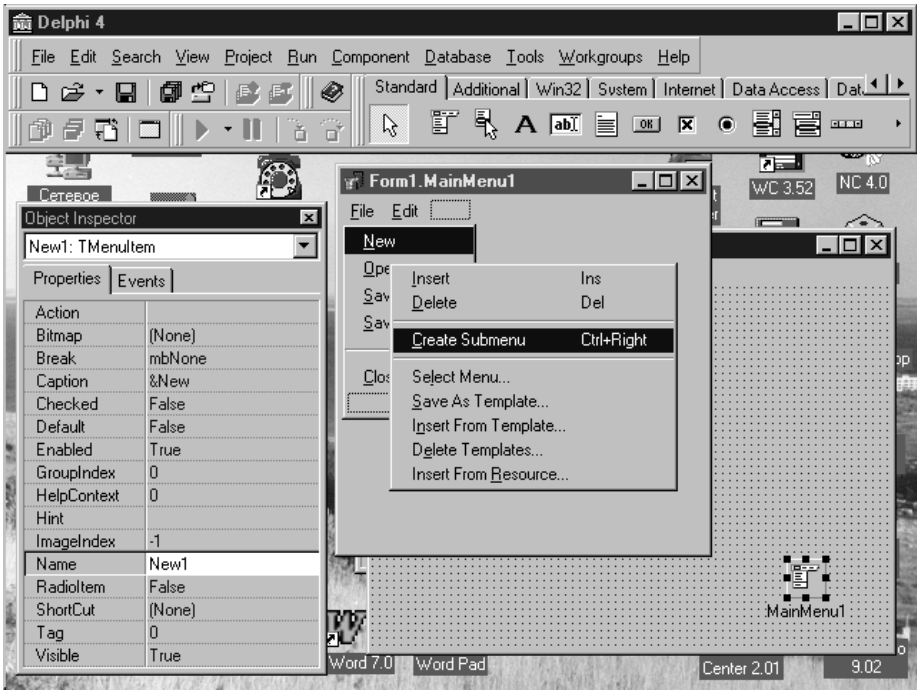


Рис.16. Проектирование подменю в главном меню (**TMainMenu**)

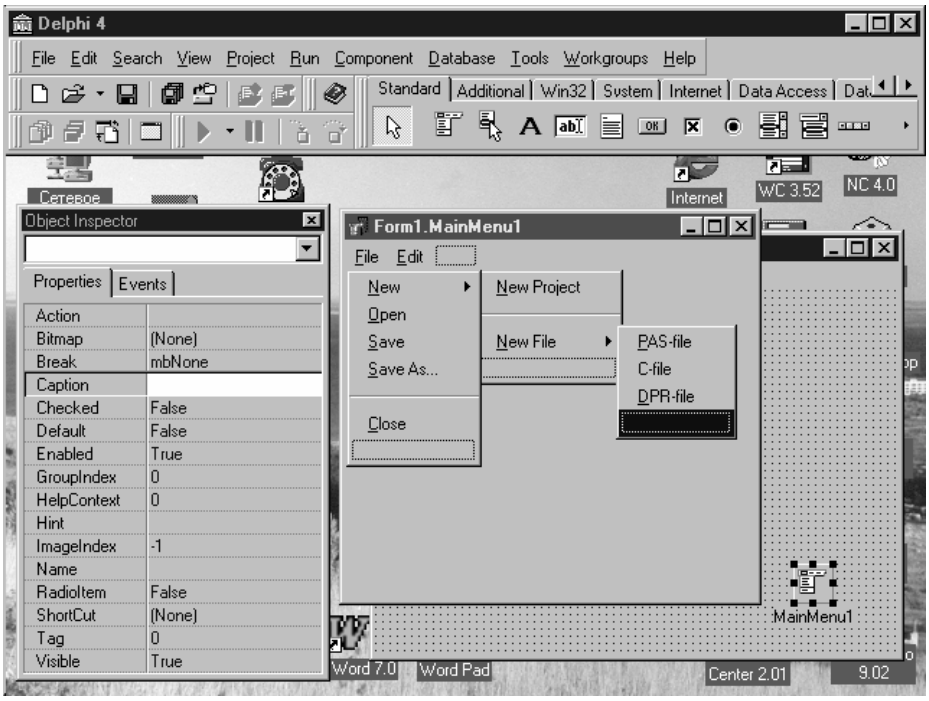


Рис.17. Проектирование подменю второго уровня в главном меню (**TMainMenu**)

Для создания вложенного меню следует щелкнуть правой клавишей 'мыши' на том пункте меню, для которого предполагается вложение и выбрать во всплывшем меню вариант **Create Submenu** (см. рис.16).

Если приложение имеет несколько форм со своими **TMainMenu**, то для упрощения работы приложения целесообразно соединить их в одно и управлять меню из главной формы, для этого у меню главной формы следует установить свойство **AutoMerge=FALSE**, а для меню присоединенных форм - **AutoMerge=TRUE**. Этот вариант незаменим при создании **MDI**-приложений и при работе с **OLE**-серверами.

6.2. КОМПОНЕНТ TPopupMenu

Для проектирования всплывающего меню следует 'положить' на форму компонент **TPopupMenu** и с помощью **Object Inspector**'а настроить свойства объекта; в пределах одной формы может быть несколько компонентов **TPopupMenu** (каждый должен быть зарегистрирован в свойстве **PopupMenu** родительского компонента, во время **RunTime** по щелчку правой кнопки 'мыши' будет вызываться то **PopUp**-меню, владельцем которого является компонент, в пределах которого произошел щелчок правой клавишей 'мыши').

Проектирование **PopUp**-меню не отличается от проектирования стандартного меню (см. рис.18).

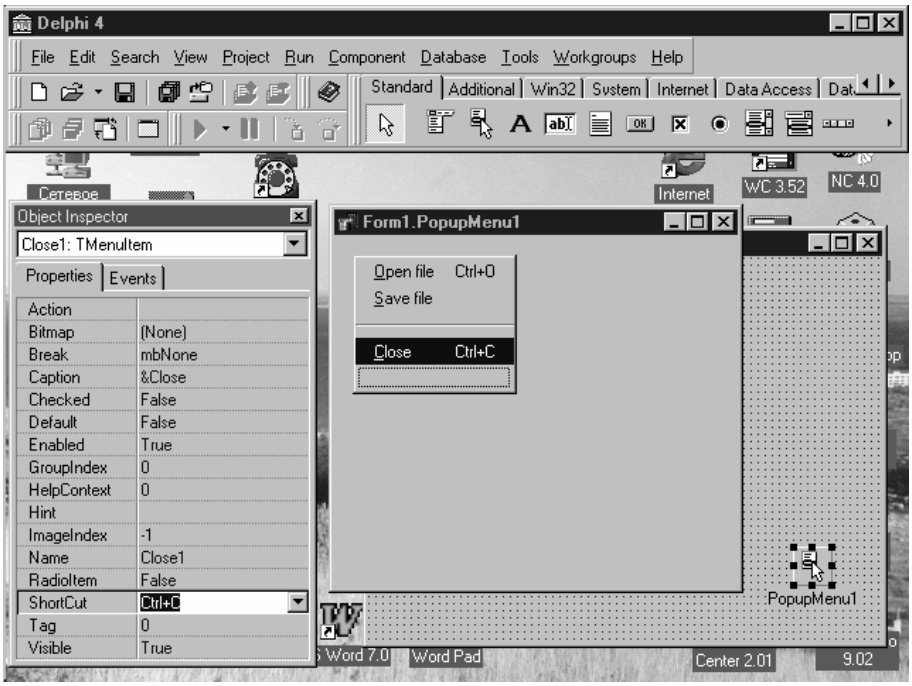


Рис.18. Проектирование всплывающего меню (**TPopupMenu**)

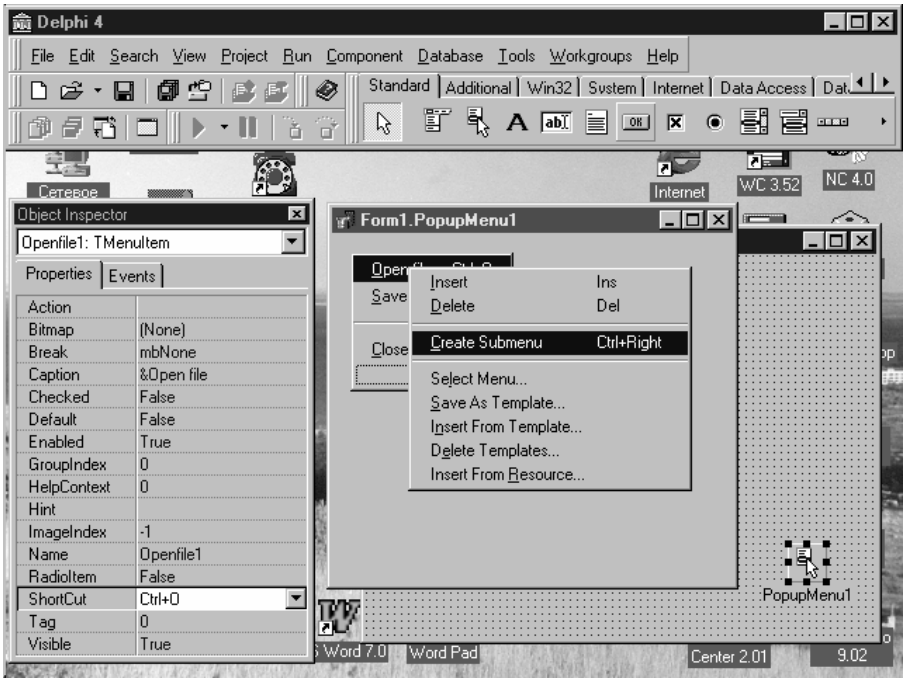


Рис.19. Проектирование подменю во всплывающем меню (**TPopupMenu**)

Вложение пунктов меню также инициируется правой клавишей 'мыши' и выбором варианта **Create Submenu** (см. рис.19).

Дальнейшие действия по проектированию вложенных пунктов меню не отличаются от описанных вышеописанных (рис.20).

С каждым пунктом меню можно связать обработчик события **OnClick**, в котором разработчик пишет функциональную часть программы. Кроме того, в момент ‘всплытия' меню возникает событие **OnPopUp**, которое разработчик может использовать по желанию.

Как главное, так и всплывающее меню может быть модифицировано и во время выполнения приложения (**RunTime**); это важное свойство доступно подготовленным **Delphi** / **C++Builder** - программистам.

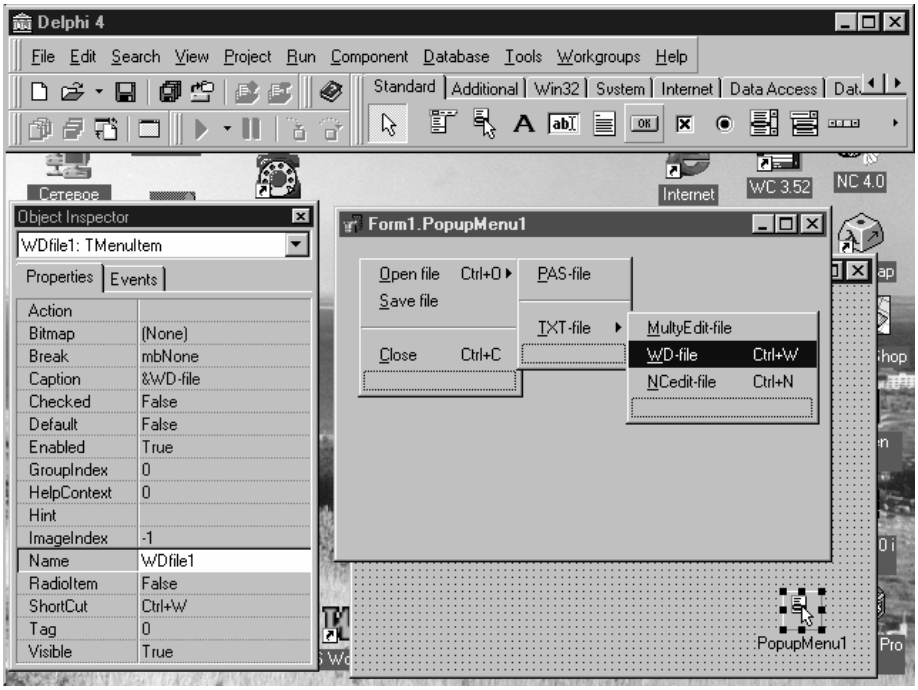


Рис.20. Проектирование подменю второго уровня во всплывающем меню (**TPopupMenu**)

7. **РИСОВАНИЕ В Delphi И КЛАСС TCanvas**

Традиционно рисование в WINDOWS реализуется чрезмерно сложно (что противоречит огромным возможностям графической оболочки). **Delphi** и **C++Builder** инкапсулирует низкоуровневые графические операции, предоставляя доступ к графическим функциям через свойство **Canvas** ('канва' для рисования), представляющее собой область окна (в случае формы), в которую можно выводить различные графические изображения (аналог дисплейного контекста для рабочей области окна).

Компоненты **TForm**, **TBitmap**, **TListBox**, **TSringList**, **TFileListBox**, **TDirectoryListBox** и другие имеют свойство **Canvas** и, значит, доступны для вывода графических изображений.

Класс **TCanvas** инкапсулирует в себе основные операции **GDI** (*Graphic Device Interface*) и позволяет программисту сосредоточиться на графических

операциях, не вдаваясь в тонкости программирования конкретного устройства представления графики.

Класс **TCanvas** обладает свойствами **Brush** ('кисть' - цвет и шаблон заполнения при графических операциях), **CopyMode** (режим копирования; по умолчанию **CopyMode=cmSrcCopy** - замещение текущего изображения), **Font**, **Pen** (тип ‘карандаша'), **PenPos** (текущая позиция 'карандаша'), **Pixels** (прямой доступ к отдельным пикселам изображения).

Например, в следующем примере в прямоугольной области (заданной левым верхним углом **X=100,Y=200** и правым нижним **X=300,Y=400**) каждый пиксел цвета **clRed** заменяется на пиксел цвета **clBlue**

```
var
  X,Y: integer;
witn DrawForm.Canvas do
begin
  for X:=100 to 300 do  { цикл по оси абсцисс }
    for Y:=200 to 400 do  { цикл по оси ординат }
      if Pixels[X,Y]=clRed  then  { если текущий пиксел красный... }
        Pixels[X,Y]:=clBlue;  { ...то сделать его синим }
end;
```

Собственно рисование реализуется методами класса **TCanvas**, здесь и ниже приведены примеры в расчете рисования на форме (или компоненте) с именем **DrawForm**.

Метод **Arc(Xl,Yl,X2,Y2,X3,Y3,X4,Y4:integer)** изображает дугу, заключенную (вписанную) в прямоугольник с левым верхним **X1,Y1** и правым нижним **X2,Y2** углами, причем начальная и конечная точки дуги суть **X3,Y3** и **X4,Y4** соответственно

```
witn DrawForm.Canvas do
begin
  Pen.Color:=clRed; { выбрать красный карандаш  }
  Pen.Width:=3; { ширина карандаша 3 пиксела  }
  Arc(100,100, 200,200,100,0,100,0); { рисовать эллипс }
end;
```

Метод **Chord(X1,Y1,X2,Y2,X3,Y3,X4,Y4:integer)** рисует соединяю-щую две точки дуги хорду (параметры соответствуют таковым метода **Arc**).

Метод **CopyRect(Dest:TRect; Canvas:TCanvas; Source:TRect)** копирует заданную канвой **Canvas** прямоугольную область **Source** в прямоугольную область **Dest** текущей канвы; режим копирования может быть предварительно задан свойством **CopyMode** (например, **DrawForm.Canvas.CopyMode:=cmPatCopy** для копирования с логической операцией **XOR** при смешении цветов).

Метод **Draw(X,Y:integer; Graphic:TGraphic)** отображает графический объект типа **Graphic** в точке с координатами **X,Y**.

Метод **FillRect(Rect:TRect)** выполняет заливку прямоугольной области **Rect** цветом, который задан значением свойства **Brush.Color**

```
witn DrawForm.Canvas do
begin
  Brush.Color: =clYellow;  { установим желтую кисть }
  FillRect(Rect(100,100, 200,200));  { залить желтым }
end;
```

Метод **FloodFill(X,Y:Integer; Color:TCoIor; FillStyle:TFillStyle)** заполняет область заданным значением свойства **Brush** цветом (заполнение начинается с точки **X,Y**). **FillStyle** задает режим заполнения - при **FillStyle=fsBorder** область заполнения ограничена цветом **Color**, при **FillStyle=fsSurface** область заполняется до тех пор, пока в не присутствует хотя бы один пиксел цветом **Color** (используется для имеющих многоцветную границу областей).

Метод **LineTo(X,Y:integer)** проводит прямую линию из текущей в точку с координатами **X,Y** (которая становится текущей и запоминается в свойстве **PenPos**).

Изменение координат текущей точки (без отрисовки линии) производится методом **MoveTo(X,Y: integer)**.

Метод **Pie(Xl,Yl, X2,Y2, X3,Y3, X4,Y4:integer)** отрисовывает сегмент эллипса (формальные параметры соответствуют таковым для метода **Arc**).

Метод **Polygon(Points:array of TPoint)** отрисовывает замкнутый (залитый) многоугольник, заданный массивом координат **Points**, а метод **Polyline(Points:array of TPoint)** отрисовывает соответствующий многогранник.

Метод **Rectangle(X1,Yl,X2,Y2:integer)** отрисовывает прямоугольник с левой верхней и правой нижней точками **X1,Y1** и **X2,Y2** соответственно.

Метод **RoundRect(X1,Yl,X2,Y2,X3,Y3:integer)** отрисовывает прямоугольник с закругленными углами (параметры **X1,Y1,X2,Y2** соответствуют таковым метода **Rectangle**, **X3** и **Y3** -размеры четвертей эллипса, отображаемых в вершинах прямоугольника).

Программист не должен забывать о настройке цвета и толщины карандаша при отрисовке вышеуказанных графических примитивов.

Метод **StretchDraw(Rect:TRect; Graphic:TGraphic)** производит масштабирование графического объекта **Graphic** в прямоугольник **Rect**, методы **TextHeight(Text:string):integer** и **TextWidth(Text:string):integer** возвращают высоту и ширину строки **Text** в пикселах (с учетом шрифта строки), метод **TextOut(X,Y:integer; Text:string)** выводит строку **Text** начиная с левой верхней точки **X,Y** (не следует забывать задать шрифт и размер оного).

Заметим, что рекомендуется после каждого достаточно зн**а**чимого изме-

нения изображения принудительно перерисовывать область вывода путем использования родительских методов **Paint** (без предварительной очистки области) или (иногда) **Repaint** (область вывода перед перерисовкой очищается).

К сожалению, подобные вышеприведенным алгоритмы очень неэффективны по затратам машинного времени (вследствие многократного вывода на экран отдельных пикселов); профессионалльный подход заключается в создании внеэкранного битового образа - например, с помощью компонента типа **TBitmap** - и 'сброс' оного на экран (режим копирования **CopyMode= cmSrcCopy**) с помощью метода **Draw**.

8. ПЕЧАТЬ В Delphi И C++Builder

Вывод на печать - вторая традиционно сложная группа операций в WINDOWS. **Delphi** и **C++Builder** предоставляют разработчику компонент **TPrinter**, однако мало упрощающий организацию вывода на печать.

Версии **Delphi** выше 1.0 и **C++Builder** имеют штатный компонент **TQuickReport**, в максимальной степени упрощающий организацию печати; см. документацию указанного компонента.

Только в сложных случаях целесообразно применять систему **Report Smith**, обладающую огромными возможностями, но требующую больших затрат памяти и медленно работающую даже на мощных ПЭВМ.

9. НЕКОТОРЫЕ ПОЛЕЗНЫЕ ФУНКЦИИ И ПРИЕМЫ ПРОГРАММИРОВАНИЯ В Delphi И C++Builder

9.1. ЧАСТО ИСПОЛЬЗУЕМЫЕ ФУНКЦИИ И ПРОЦЕДУРЫ

При работе с **Delphi** часто приходится использовать некоторые функции, непривычные даже для знатоков языка **Pascal**; ниже приведены советы по применению этих функций.

При отладке часто приходится выводить в компонент **TLabel** числовые значения, в то же время свойство **Caption** принимает только строковые значения. Ниже показано применение функций простейшего (без возможности указания числа позиций для целого и вещественного) форматирования целых и вещественных величин для преобразования в строку

```
Label1.Caption :=  IntToStr(NumberOfCars) + ' : ' +
                   FloatToStr(SpeedOfCar);
```

Существуют реализующие обратное преобразование функции **StrToInt** и **StrToFloat**. Пример несколько более сложного форматирования показан ранее - см. раздел 5.6 данной работы. Наибольшей же гибкостью обладают

функции форматирования и преобразования (обычно включают в своем имени слово '**Format**). Например, функция

```
Format(const Fmt: string, const Args: array of consts): string;
```

возвращает строку, представляющую собой отформатированные по шаблону **Fmt** (в основном используются **C**-шаблоны форматирования) ряд переменных **Args**. Например, в следующем примере в поле **LabelOut** будут выведены - текст 'Значение...', расположенное в 4 позициях целое I и расположенное в 10 позициях (с тремя знаками после запятой) вещественное число R.

```
NumbI: integer;
NumbR: real;
. . . . . .
LabelOut.Caption:=Format('Значение I=%4d, R=%10.3f, [NumbI, NumbR]);
```

C++Builder. Соответственно

```
int NumbI;
float NumbR;
. . . . . .
char buffer_1 [100];  // C-строка из 100 символов
sprintf(buffer_1, "Значение I=%4d, R=%10.3f", NumbI,NumbR);
LabelOut->Caption =buffer_1;
//  Delphi-подобная функция Format также доступна, однако
// при передаче в нее форматируемых параметров необходимо
// преобразовывать их к типу 'array of const' (TVarRec) - см. образец ниже
// TVarRec str[3] = {"Значение", NumbI,NumbR"};
// Label1->Caption=Format("%s I=%4d , R=%10.3f", str,3);
```

Естественно, существует ряд функций обратного преобразования ('строка число'). Для более подробного ознакомления рекомендуется литература [5,8,10,11,13].

Большинство авктивизируемых событиями процедур передают вызывающей подпрограмме значение параметра **Sender**, указывающего на объект, вызвавший событие. Во многих случаях (например, при наличии множества выполняющих близкие функции кнопок) нет необходимости связывать собственную процедуру-обработчик с каждым Control'ом. Нижеприведенный Pascal-код иллюстрирует возможность идентификации (по имени) компонента, вызвавшего процедуру:

```
...
if (Sender as TComponent).Name = 'Button_01' then
  { обработка нажатия кнопки с именем Button_01 }
else
if (Sender as TComponent).Name = 'Button_02' then
```

{ обработка нажатия кнопки с именем Button_02 }
... { и так далее }

Дополнительно целям обмена информацией служит свойство **Tag**, имеющееся у большинства компонентов. Объектно-ориентированная структура **Delphi / C_Builder / Kylix** позволяет совершать буквально чудеса программирования (например, описанная в [9] возможность вызовов компонентных методов дальних предков определенного класса).

В языке **Object Pascal 8.0** существуют два типа строк - в стиле **Pascal**'я (объект имеет длину до 256 байт, в нулевом байте записана длина текстовой части объекта в байтах - т.е. допустимо хранить не более 255 символов, тип **string**) и в стиле **C** (длина строки не ограничена, признаком конца строки является нуль, тип **PChar**); различные функции (особенно функции WINDOWS API) оперируют с различными типами строк. Определены функции конвертации различных типов строк - например, функции **StrPCopy** и **StrPas**.

Полезны функции работы с переменными типа даты и времени - **DayOfWeek** (возвращает номер текущего дня недели в диапазоне $1 \div 7$), **Date** (возвращает текущую дату), **Time** (возвращает текущее время), **Now** (возвращает текущие дату и время), **DateToStr**, **TimeToStr**, **DateTimeToStr** (конвертируют соответствующие величины из внутреннего представления в строку текста), **StrToDate**, **StrToTime**, **StrToDateTime** (выполняют обратное преобразование) и др.

Из функций, работающих с файлами, интересны **FileOpen**, **FileCreate**, **FileRead**, **FileWrite**, **FileSeek**, **FileClose**, **RenameFile**, **DeleteFile** (открывает, создает, читает, записывает, позиционирует указатель, закрывает, переименовывает и уничтожает файл соответственно), **FileAge** и **FileExist** (возвращает дату и время создания файла и проверяет существование файла), **FindFirst** и **FindNext** (осуществляют поиск файлов по маске), **ChangeFileExt**, **ExtractFilePath**, **ExtractFileName**, **ExtractFileExt**, **ExpandFileName**, **FileSearch** (изменяет расширение имени файла, извлекает из строки с полным именем файла путь к файлу, извлекает из строки имя файла, извлекает из строки расширение имени файла, возвращает полное имя файла, производит поиск файла соответственно), **DiskFree**, **DiskSize** (возвращает количество свободного места на диске и размер диска в байтах).

Для работы с большими объектами служат функции **AllocMem**, **ReAllocMem**, **MemAlloc** и **FreeMem** (выделяет и обнуляет блок памяти, изменяет размер блока памяти, выделяет блок памяти размером более 64К байт и освобождает память).

Подробности применения этих функций см. в системе контекстной помощи **Delphi** или в книге [8]; хороший обзор используемых **C++Builder**'ом функций (включая часто используемые функции WINDOWS API) приведен в работе [13].

9.2. ПРИЕМЫ РАБОТЫ С КОМАНДНОЙ СТРОКОЙ И ПРОЦЕССАМИ-ПОТОМКАМИ

Стандарты языка **Object Pascal 8.0** системы **Delphi** и языка **C++** пакета **C++Builder** позволяют создавать программное обеспечение не только прикладного, но и системного уровня; некоторые из этих возможностей будут продемонстрированы ниже на примерах обработки параметров командной строки и управления процессами-потомками.

Число параметров командной строки может быть прочитано из переменной **ParamCount,** причем д**о**ступ к i-тому параметру командной строки осуществляется как **ParamStr(i)**, где **i=0÷ParamCount-1**.

Нижеследующий фрагмент **Pascal**-кода заносит в компонент **Memo1** содержимое параметров командной строки

```
Memo1.Clear; { очистка  Memo1 }
{ заполнение списка  Memo1  параметрами командной строки }
for i := 1 to ParamCount do
  Memo1.Lines.Add(ParamStr(I));
```

C++Builder. Аналогичный пример **C**-кода приведен ниже

```
Memo1->Clear(); // очистка  Memo1
// заполнение списка  Memo1  параметрами командной строки
for (i=0; i <= ParamCount(); i++)
  Memo1->Lines->Add(ParamStr(i));

Memo2->Clear();// очистка  Memo2
// заполнение списка  Memo2  переменными среды Windows
i=0;
 while (_environ[i])
  Memo2->Lines->Add(_environ[i++]);
```

Процесс в WINDOWS'9x и WINDOWS'NT запускается с помощью системной функции **CreateProcess**; с помощью этой функции можно запустить как 32-, так и 16-тиразрядные приложения WINDOWS, а также программы MS-DOS и 16-тиразрядные консольные приложения OS/2. Для инициализации процесса-потомка может быть использована нижеприведенная **Pascal**-процедура **RunExternal**, основой которой является как раз функция **CreateProcess**

```
procedure TForm1.RunExternal(CommandLine: string;
                             RuleParent,Priority: byte;
                             RuleMessage: boolean);
{  пытается стартовать процесс-потомок согласно командной строке
```

```
CommandLine
   при RuleParent=0 процесс-родитель ждет окончания работы потомка, при
     этом   позволяя работать другим WINDOWS-приложениям (т.е. 'спит')
   при RuleParent=1 процесс-родитель продолжает работать вместе с потом-
     ком
   при всех других значениях RuleParent процесс-родитель   после запуска
     процесса-потомка завершается
   Priority=0/1/2/3  соответствует  приоритетам  запускаемого   приложения
     REALTIME / HIGH / NORMAL / IDLE соответственно (все другие значения
     Priority соответствуют IDLE)
   при RuleMessage=TRUE выдается сообщение об ошибках  }
const
  CRLF = #13#10;
var
  si: STARTUPINFO; { структура определения внешнего вида
                           окна процесса-потомка }
  pi: PROCESS_INFORMATION; { структура хранения идентификаторов
                                    и системных номеров созданного процесса и
                                    его главной задачи }
  zString: array[0..255] of Char;
  dwCreationFlag,dwExitCode: DWORD;
  out: boolean;
begin

  FillChar(si, sizeof(si), 0); { обнулить структуру  si  }
  si.cb:=sizeof(STARTUPINFO);  { заполним поле  cb  структуры  si  }

  case Priority of   {  настройка приоритета процесса-потомка  }
    0: dwCreationFlag:=REALTIME_PRIORITY_CLASS;
    1: dwCreationFlag:=HIGH_PRIORITY_CLASS;
    2: dwCreationFlag:=NORMAL_PRIORITY_CLASS;
    else
      dwCreationFlag:=IDLE_PRIORITY_CLASS;
  end;

  out:=CreateProcess(NIL,StrPCopy(zString,CommandLine),NIL,NIL,false,
                           dwCreationFlag,NIL,NIL,si,pi);

  if out = false then  {  если старт неудачен... out=false  }
  begin
    if RuleMessage then { задан режим выдачи сообщений об ошибках  }
      MessageDlg('Извините, выполнение' + CRLF + CRLF +
                       UpperCase(CommandLine) + CRLF + CRLF +
                       'невозможно...  (ошибка '+IntToStr(GetLastError()) + ')',
                       mtError,
                       [mbOk], 0);
     exit;
   end; { конец  IF out = false  }
```

```
   if out = true then { если старт удачен... out=true }
   begin
     if RuleParent = 0 then { если родитель должен ждать окончания
                              работы потомка  }
     begin
       CloseHandle(pi.hThread); { хэндл потока не нужен - удаляем }
       { начинаем бесконечный цикл ожидания... }
       if WaitForSingleObject(pi.hProcess, INFINITE) <> WAIT_FAILED then
       begin
       GetExitCodeProcess(pi.hProcess, dwExitCode); { если ошибка функции
                                                      ожидания... }
       if RuleMessage then { если задан режим выдачи сообщений
                             об ошибках }
       if dwExitCode <> WAIT_OBJECT_0 then {  WAIT_OBJECT_0 = естественное
                                              завершение процесса }
       MessageDlg('Извините, процесс ' + CRLF + CRLF +
                  UpperCase(CommandLine) + CRLF + CRLF +
                  'закончен с ошибкой '+IntToStr(dwExitCode) + CRLF,
                  mtError,  [mbOk], 0);
       CloseHandle(pi.hProcess);  { освобождаем хэндл процесса  }
       end; { конец IF WaitForSingleObject... }
   end; { конеw IF RuleParent = 0 }

     if RuleParent = 1 then { родитель не должен ждать окончания
                              работы процесса-потомка  }
   exit; { выход - ничего не делая }

   if (RuleParent <> 0) and { родитель завершается }
      (RuleParent <> 1) then
      Application.Terminate; { закончить родительский процесс }

  end; { конец IF out=true }

end; { конец процедуры RunExternal }
```

C++Builder. Полностью функциональный **C**-аналог вышеприведенной **Pascal**-процедуры **RunExternal** приведен ниже

```
void
__fastcall TForm1::RunExternal(char* CommandLine,
                               byte RuleParent,
                               byte Priority,
                               bool RuleMessage)
{
  STARTUPINFO si;
  PROCESS_INFORMATION pi;
  DWORD dwCreationFlag,dwExitCode;
  bool out;
```

```
memset(&si, 0, sizeof(STARTUPINFO)); // обнулить структуру   si
si.cb=sizeof(STARTUPINFO);  // заполним поле cb  структуры  si

switch (Priority)  // настройка приоритета процесса-потомка
{
  case 0: dwCreationFlag=REALTIME_PRIORITY_CLASS;
             break;
  case 1: dwCreationFlag=HIGH_PRIORITY_CLASS;
             break;
  case 2: dwCreationFlag=NORMAL_PRIORITY_CLASS;
             break;
  default:
             dwCreationFlag=IDLE_PRIORITY_CLASS;
}

out=CreateProcess(NULL,CommandLine,NULL,NULL,false,
                          dwCreationFlag,NULL,NULL,&si,&pi);

if (!out) // если старт неудачен... out=false
{
  if (RuleMessage)  // если задан режим выдачи сообщений об ошибках
    MessageDlg("Извините, выполнение\n\r\n\r" +
                       UpperCase(CommandLine)  +
                       "\n\r\n\rневозможно...  (ошибка " +
                       IntToStr(GetLastError()) + ")",
                       mtError, TMsgDlgButtons() << mbYes, 0);
  return;
} // конец IF out = false

if (out) // если старт удачен... out=true
{
  if (RuleParent == 0) // если родитель должен ждать
                            // окончания работы потомка
   {
    CloseHandle(pi.hThread); // хэндл потока уже не нужен - удаляем
    // начинаем бесконечный цикл ожидания...
    if (WaitForSingleObject(pi.hProcess, INFINITE) != WAIT_FAILED)
    {
      GetExitCodeProcess(pi.hProcess, &dwExitCode);  // если
                                                 // ошибка функции  ожидания...
      if (RuleMessage) // если задан режим выдачи сообщений об ошибках
      if (dwExitCode != WAIT_OBJECT_0) // WAIT_OBJECT_0 = естественное
                                                  // завершение процесса
      MessageDlg("Извините, процесс\n\r\n\r" +
                        UpperCase(CommandLine) +
                        "\n\r\n\rзакончен с ошибкой " +
                        IntToStr(dwExitCode) + "\n\r",
                        mtError, TMsgDlgButtons() << mbOK, 0);
```

```
      CloseHandle(pi.hProcess); // освобождаем хэндл процесса
    } // конец IF WaitForSingleObject...
  } // конец IF RuleParent = 0

  if (RuleParent == 1) // родитель не должен ждать окончания
                       // работы потомка
    return; // выход - ничего не делая

  if ((RuleParent != 0) &&  // родитель завершается
      (RuleParent != 1))
    Application->Terminate(); // закончить родительский процесс

 } // конец IF out=true

} // конец процедуры RunExternal
```

Приложение-родитель имеет возможность контролировать работу приложения-потомка (вплоть до приостановления, полного останова или возобновления выполнения потомка) с помощью WINDOWS API функций **SuspendThread**, **TerminateProcess** или **ResumeThread** соответственно; в качестве формальных параметров этим функциям передается идентификатор управляемой задачи.

Подробнее рекомендуется методическое руководство ‘Технология программирования больших программных комплексов’ того же автора.

9.3. СОЗДАНИЕ ИНТЕРФЕЙСА, НЕЗАВИСИМОГО ОТ РАЗМЕРОВ ОКНА

Взаимное расположение интерфейсных элементов на форме изменяется при изменении размеров формы (например, при ее максимизации). Проблема часто возникает также при эксплуатации приложения на ПЭВМ, снабженной дисплеем, отличным (по разрешающей способности) от дисплея машины разработчика.

Задача сохранения взаимного расположения компонентов на форме решается просто. В большинстве случае для этого используется связанный с событием **OnResize** обработчик, в котором явно задается взаимное положение каждого компонента на форме; процедура активизируется во время **RunTime**.

В качестве примера рассмотрим процедуру-обработчик, поддерживающую размер компонента **Panel1** в половину размеров формы **Form_1** и положение его же в центре оной

```
procedure TForm_1.FormResize(Sender: TObject);
begin
  Panel1.Width := Form_1.Width  div  2;
  Panel1.Height:=Form_1.Height  div  2;
```

```
  Panel1.Left := (Form_1.Width - Panel1.Width) div 2;
  Panel1.Top := (Form_1.Height - Panel1.Helght) div 2;
end;
```

Второй пример демонстрирует ограничение размеров окна (окно не может превысить размеров экрана текущего дисплея)

```
procedure TForm_1.FormReslze(Sender: TObject);
begin
  if Form_1.Width  >  Screen.Width then
    Form_1.Width := Screen.Width;
  if Form_1.Height  >  Screen.Height then
    Form_1.Height := Screen.Height;
end;
```

Среди компонентов третьих фирм-разработчиков встречаются компоненты, автоматически масштабирующиеся при изменении размеров формы (например, известен близкий к **TPanel** компонент, масштабирующий все свои дочерние компоненты).

В последних версиях систем **Delphi / C++Builder** практически каждый визуальный компонент имеет свойство BiDiMode, позволяющее ‘привязать’ данный Control относительно левой, правой, верхней, нижней (или нескольким) сторонам родительского компонента.

9.4. **ОБРАБОТКА ОШИБОК И ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ**

При функционировании реальных приложений часто возникают проблемы обработки ошибок (например, ошибок ввода, обработки файлов etc). Традиционный (включающий десятки/сотни блоков **if... then... else**) способ весьма неэффективен в сложных программах. В **Delphi / C++Builder** введены специальные расширения языка, позволяющие обрабатывать любые исключительные (как с точки зрения ЭВМ, так и пользователя) ситуации.

Слежение за исключительными ситуациями (ИСС) достигается использованием блоков **try/except** и **try/finally** (для стандарта языка **Object Pascal 8.0**).

Например, 'ловушка' ошибки преобразования при вводе в компоненте **TEdit** вещественного числа в виде строки реализуется следующим образом

```
var
  Speed: real;
. . . . . .
try  { начало блока слежения за ИСС }
   Speed:=StrToReal(SpeedEdit.Text);
except { выполняется при любой ошибке в блоке }
   MessageDlg('0шибка ввода значения скорости...',
```

```
                mtWarning, [mbOk], 0);
end; { конец блока  try/except }
```

C++Builder. В C++ вместо конструкции **try/except** используется блок **try/catch**; при этом блок **catch** выполняется при наличии ИСС в предшествующем блоке **try** согласно нижеприведенному скелетному коду

```
try  // начало блока слежения за ИСС
//
// ... любой код, могущий сгенерировать исключение ...
//
catch(T Z) // выполняется при возникновении исключения Z
           // типа T в предшествующем try-блоке
//
//  ... необходимая обработка ИСС ...
//
catch(...) // выполняется при любой ошибке в
          //  предшествующем блоке
//
// ... необходимая обработка ИСС ...
//
```

При использовании блока **try/finally** следующие за ключевым словом **finally** операторы выполняются в обязательном порядке после возникновения ИСС (важно в определенных случаях, например, для освобождения ранее выделенных блоков памяти)

```
try  { включить отслеживание ИСС }
   Operator'1;
   Operator2;
   Operator3;
finally { выполняется всегда !  }
   Operator4;
end; { конец   try / finally }
```

Имеется возможность определять реакции на определенные типы ИС, что реализуется обработчиками ИСС (объектами класса **Exception**, причем программист может создавать - кроме имеющихся базовых - объекты этого класса для обработки ожидаемых ИСС).

Например, в следующем примере (с помощью конструкции **on... do... else**) отслеживаются стандартные ИСС **EZeroDivide**, **EOutOfMemory** и **EInvalidPointer**

```
var
  RealVal: real;
  IntVal: integer;
```

```
. . . . . .
try  { начало блока try / except }
  RealVal :=10 / IntVal;
. . . . . .
... другие операторы ...
. . . . . .
except
on EZeroDivide do { деление на нуль }
  MessageDlg('Исключительная ситуация EZeroDivide',
                  mtWarning, [mbOk], 0);
on EOutOfMemory do { недостаточно памяти }
  MessageDlg('Исключительная ситуация EOutOfMemory',
                  mtWarning, [mbOk], 0);
  on EInvalidPointer do { недопустимая операция с указателем }
MessageDlg('Исключительная ситуация EInvalidPointer',
                 mtWarning, [mbOk], 0);
else { другая исключительная ситуация  }
  MessageDlg('Неизвестная исключительная ситуация',
                  mtWarning, [mbOk], O);
end; { конец   try / except }
```

Программист может принудительно возбудить любую ИСС с помощью функции **raise/throw** (**Delphi** / **C++Builder** соответственно).

9.5. **ШАБЛОНЫ ПРИЛОЖЕНИЙ И ФОРМ**

При создании приложений часто приходится использовать похожие скелетные структуры (например, скелеты **MDI**- и **SDI**-приложений). В ИС **Delphi** / **C++Builder** имеется возможность использовать шаблоны приложений.

Пользователь имеет возможность выбрать в ответ на запрос создания нового проекта выбор любого из шаблонов (используя последовательность **Options|Environment|Preferences** и пометив кнопку **Use on New Project** в группе **Gallery**).

В список предопределенных шаблонов входят - **Blank Project** (отказ от шаблона), **MDI Application** (две формы и три модуля для создания стандартного **MDI**-приложения), **SDI Application** (стандартное **SDI**-приложение), **CRT Application** (простейшее приложение для вывода текста в окно программы) и другие.

После выбора скелета программист имеет возможность переопределить/добавить компоненты в приложения и сохранить (переработанные) шаблоны в галлерею с помощью **Save|Save As Template** (или используя правую кнопку 'мыши').

Delphi поддерживает также галлерею форм (включается через **Options|Environment|Preferences** и пометив кнопку **Use on New Form** в группе **Gallery**).

Для использования предопределенных форм из галлереи следует выполнить **File|Remove File** для удаления пустой формы из проекта и выбрать форму из галлереи путем **File|New Form** и выбора нужной формы из списка. Для сохранения собственной разработанной формы в галлерее следует вызвать окно **Save Form Template** (путем нажатия правой кнопки 'мыши' и выбора варианта **Save As Template**), выбрать из списка сохраняемую в виде шаблона форму и ввести ее краткое описание. Классический пример шаблона формы - форма **About** с информацией о данном приложении и фирме-разработчике.

10. **ПРИМЕР СОЗДАНИЯ РЕАЛЬНОГО ПРИЛОЖЕНИЯ В Delphi**

Приведем пример разработки несложного (имеющего всего одну форму) приложения - утилиты просмотра графических файлов, создаваемого (достаточно квалифицированным разработчиком) с помощью **Delphi** за несколько минут.

На рис.21 приведена копия экрана дисплея в **DesignTime**; на форме расположены компоненты **TPanel** и (дочерний по отношению к **TPanel** компонент **TImage**, причем свойство **Align** компонента **TImage** установлено в **alClient**, что гарантирует 'заполнение' компонентом **TImage** всего пространства компонента **TPanel** при любом изменении размеров последнего) и три компонента **TBitBtn** (кнопки), для выбора файла изображения присутствует **TOpenDialog**. Ниже приведен текст единственного **Pascal**-модуля **UNIT_1.PAS** со спроектированными процедурами. Заметим, что разработчиком написаны только тела процедур **BitBtn1Click**, **BitBtn2Click** и **BitBtn3Click** (обработчики нажатий на кнопки **Файл**, **Масштаб** и **Выход** соответственно); все остальное создано **Delphi** в период проектирования приложения.

```
Unit Unit_1;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics,
  Controls, Forms,    Dialogs, StdCtrls, Buttons, ExtCtrls;

type TForm_1 = class(TForm)
  Panel1: TPanel;
  Image1: TImage;
  BitBtn1:TBitBtn;
  BitBtn2: TBItBtn;
  BitBtn3: TBitBtn;
  OpenDialog1: TOpenDialog;
```

```
  procedure BitBtn1Click(Sender: TObject);
  procedure BitBtn2Click(Sender: TObject);
  procedure BitBtn3Click(Sender: TObject);

private
  { Private declarations }
public
  { Public declarations }
end;

var
  Form_1: TForm_1;

implementation

{$R *.DFM}

procedure TForm_1. BitBtn1Click(Sender: TObject);
{  выбирает и загружает файл для просмотра  }
begin
  if OpenDialog1.Execute then
    Image1. Picture.LoadFromFile(OpenDialog1.FileName);
end;

procedure TForm_1.BitBtn2Click(Sender: TObject);
{  включает и выключает режим масштабирования изображения }
begin
  Image1.Stretch := not  Image1.Stretch;
end;

procedure TForm_1.BitBtn3Click(Sender: TObject);
{ заканчивает работу }
begin
  Close;
end;

end.
```

Рис.22 представляет копию окна приложения в режиме масштабирования (изображение 'растянуто' согласно размерам окна вывода), на рис.23 то же самое без масштабирования (виден только верхний левый угол изображения).

Сторонний разработчик может сколь угодно усложнить данную программу (чему имеется немало причин).

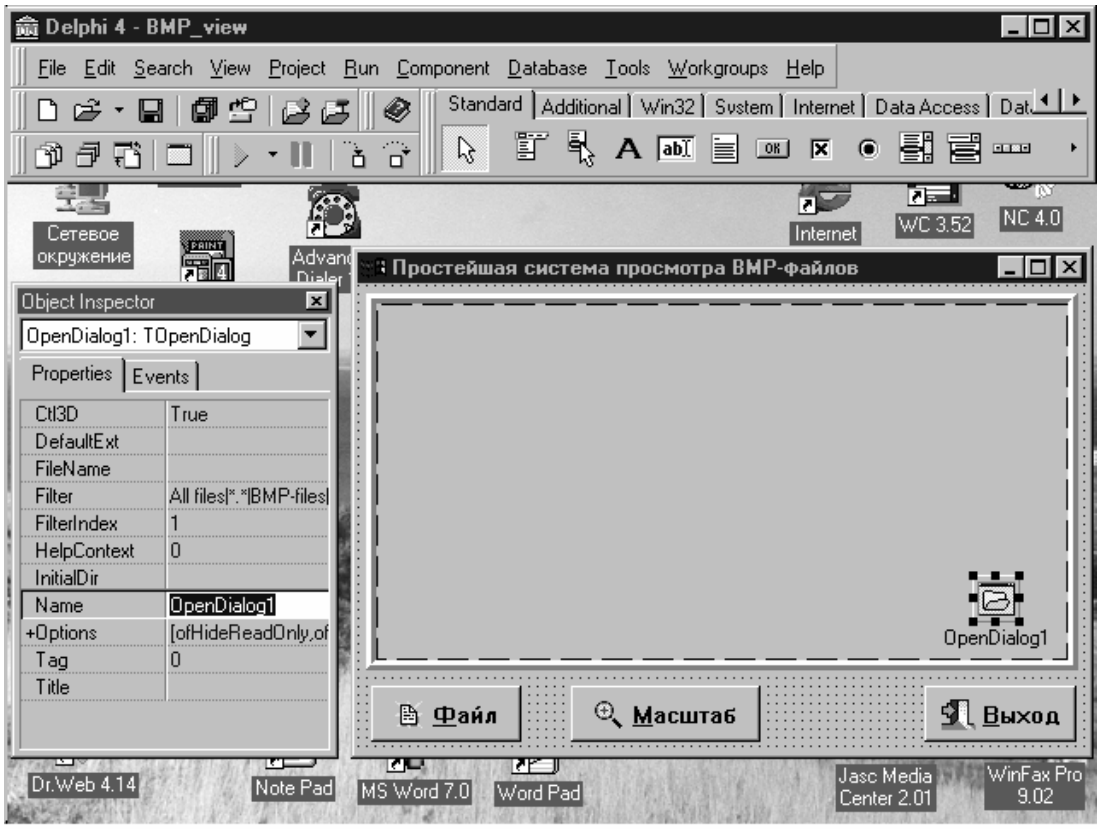


Рис.21. Копия экрана дисплея в период проектирования приложения

11. ВОЗМОЖНОСТЬ ПРЯМЫХ СИСТЕМНЫХ ВЫЗОВОВ WINDOWS

Системы **Delphi** и **C++Builder** (как, впрочем, и все подобные системы) является фактически всего лишь надстройкой над WINDOWS, скрывающей (инкапсулирующей) многие (в большинстве случаев несущественные) тонкости обращения к системным функциям WINDOWS (имеется в виду WINDOWS API - *Application Programming Interface*, [4]). Однако разумная система такого рода должна допускать прямое обращение к указанным функциям, что необходимо для продвинутых разработчиков и нестандартной работы с внешними устройствами ПЭВМ.

Первый (классический) пример использования функций WINDOWS API (необходимо включить в список **USES** модуль **MMSystem**)

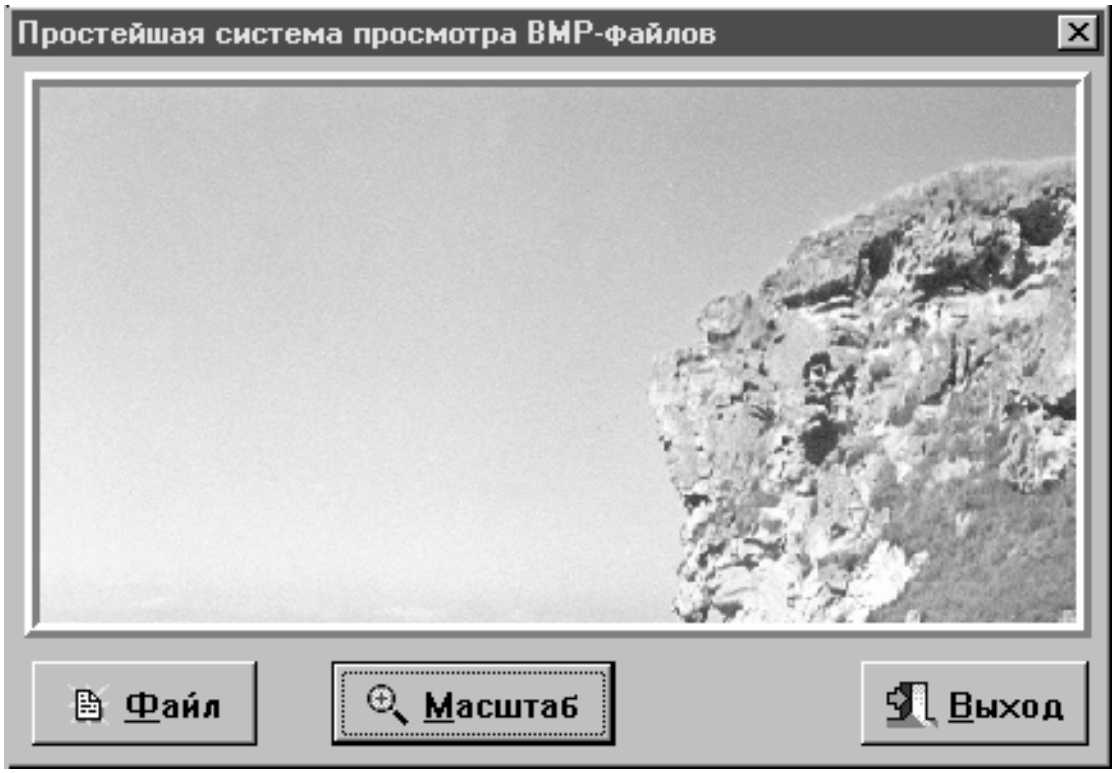


Рис.23. Окно утилиты просмотра графических файлов при отключенном режиме масштабирования

Рис.22. Окно утилиты просмотра графических файлов при включенном режиме масштабирования

```
procedure TForm1.BitBtn7Click(Sender: TObject);
{ воспроизвести звук из заданного файла }
begin
  sndPlaySound('c:\wmdows\chord.wav', 0); // проиграть музыку из файла
end;
```

Ниже приведен конкретный пример работы с системной для WINDOWS API функцией **GetDeviceCaps**. Принадлежащая **Delphi**-форме **Form1** **Pascal**-процедура **GetInformAboutPrinter** посредством вызова компонентной функции **Handle** компонента **Printer** 'берет' дескриптор устройства и вызывает WINDOWS API функцию **GetDeviceCaps** (передавая дескриптор принтера в качестве первого и соответствующую требуемому запросу константу в качестве второго фактических параметров вызова).

Здесь используются следующие константы - **TECHNOLOGY** для определения типа устройства, **RASTERCAPS** - для определения возможности- данного принтера обрабатывать текст и графику и другие (установленные в возвращаемом функцией **GetDeviceCaps** целом биты соответствуют определенным возможностям выбранного принтера); вывод данных осуществляется в стандартное вызываемое **Delphi**-функцией **MessageDlg** окно.

```
procedure TForm1.GetInformAboutPrinter(Sender: TObject);
const
  CRLF=#13#10;
var
  PrDC: HDC;
  PrInfo: integer;
  StrInfo1, StrInfo2: string;
begin
PrDC := Printer.Handle;  { возьмем дескриптор принтера }
PrInfo := GetDeviceCaps(PrDC, TECHNOLOGY);
case PrInfo of { какой тип устройства ? }
  DT_PLOTTER:        StrInfo1 :=  ‘vector plotter’;
  DT_RASDISPLAY:   StrInfo1 := ‘raster display";
  DT RASPRINTER:    StrInfo1 := ‘raster printer';
  DT_RASCAMERA:   StrInfo1 := 'raster camera';
  DT_CHARSTREAM: StrInfo1 := 'character stream';
  DT_METAFILE:        StrInfo1 := ‘metafile’;
  DT_DISPFILE:         StrInfo1 := 'display file';
else
  StrInfo1 := ‘? , kode = ‘ + IntToStr(PrInfo);
end; { конец CASE }

PrInfo := GetDeviceCaps(PrDC, RASTERCAPS);
{  уточним возможности устройства }
StrInfo2 := ‘? , kode = ‘ + IntToStr(PrInfo);
```

```
If (Prinfo and RC_BANDING) = 0 then
  StrInfo2 := 'supports banding'
else if (PrInfo and RC_BIGFONT) = 0 then
  StrInfo2 := 'supports fonts larger than 64K'
else if (PrInfo and RC_BITBLT) = 0 then
  Strinfo2 := 'transfers bitmaps'
else if (PrInfo and RC_BITMAP64) = 0 then
  StrInfo2 := 'supports bitmaps larger than 64K'
else if (PrInfo and RC_DEVBITS) = 0 then
  StrInfo2 := 'supports device bitmaps'
else if (PrInfo and RC_DI_BITMAP) = 0 then
  StrInfo2 := 'supports the SetDIBits and GetDIBits functions'
else if (PrInfo and RC_DIBTODEV) = 0 then
  StrInfo2 := 'supports the SetDIBItsToDevice function'
else if (Prinfo and RC_FLOODFILL) = 0 then
  StrInfo2 := 'performs flood fills'
else if (Prinfo and RC_GDI20_OUTPUT) = 0 then
  StrInfo2 := 'supports Windows version 2.0 features'
else if (Prinfo and RC_GDI20_STATE) = 0 then
  StrInfo2 := 'includes a state block in the device context'
{else if (Prinfo and RC_NONE) = 0 then
  StrInfo2 := 'supports no raster operations' }
else if (Prinfo and RC_OP_DX_OUTPUT) = 0 then
  StrInfo2 := 'supports dev opaque and DX array'
else if (Prinfo and RC_PALETTE) = 0 then
  StrInfo2 := "specifies a palette-based device"
else if (Prinfo and RC_SAVEBITMAP) = 0 then
  StrInfo2 := 'saves bitmaps locally"
else if (Prinfo and RC_SCALING) = 0 then
  StrInfo2 := "supports scaling"
else if (Prinfo and RC_STRETCHBLT) = 0 then
  StrInfo2 := 'supports the StretchBIt function'
else if (Prinfo and RC_STRETCHDIB) = 0 then
  StrInfo2 := 'supports the StretchDIBits function';
  MessageDlg('Лист ПРИНТЕРа : ' +
                    'w=' + IntToStr(Printer.PageWidth) +
                    '; h = ' + IntToStr(PrInter.PageHeight) +
                    CRLF + CRLF +
                    'ИЗОБРАЖЕНИЕ:'+
                    'w=' +IntToStr(Image1.PIcture.Width) +
                    ';h=' +IntToStr(Image1.Picture.Height) +
                    CRLF + CRLF +
                    'ТИП УС-ВА  ВЫВОДА - • + StrInfoI + CRLF +
                    'ВОЗМОЖНОСТИ -'   + StrInfo2 + CRLF +
                    'ВЕРСИЯ ДРАЙВЕРА -' +
                    IntToStr(GetDevIceCaps(PrDC,Driverversion)) + CRLF +
                    'БИТОВ НА ПИКСЕЛ-' +  IntToStr(GetDeviceCaps(PrDC,BitsPixel))
                      + CRLF +
                    'ЧИСЛО ЦВЕТОВ - •    +
```

```
            IntToStr(GetDeviceCaps(PrDC,NumColors))  + CRLF +
          'ЦВЕТОВЫХ ПЛАНОВ-• + IntToStr(GetDeviceCaps(PrDC,Planes)),
          mtInformation, [mbOk], 0);
end;
```

Обычно такие обращения доступны только квалифицированным пользователям, достаточно ознакомленным с возможностями системной библиотеки WINDOWS (сре**д**ы **Delphi** и **C++Builder** включают систему контекстной помощи по WINDOWS API).

12. ИСПОЛЬЗОВАНИЕ КОМПИЛЯТОРА С КОМАНДНОЙ СТРОКОЙ

Комплект **Delphi** / **C++Builder** включает компилятор с командной строкой (именуемый **DCC32.EXE** для **Delphi**, **BCC32.EXE** для **C++Builder**), функционирующий в среде MS-DOS. Использование этого компилятора оправдано в случае необходимости потоковой компиляции (перекомпиляции) больших (включающих несколько исполнимых файлов, включая DLL-файлы) проектов. Простой пример - изменение текста (или оформления) формы **ABOUT** требует перекомпиляции всех EXE-файлов, включающих данную **ABOUT**-форму. Нерационально пользоваться указанным пакетным компилятором с целью отладки программ - из-за отсутствия интегрированной среды процесс отладки будет более чем затруднен (хотя и принципиально возможен).

Приведем пример простого **BAT**-файла, принимающий имя проекта в качестве первого параметра командной строки (имя файла без точки и расширения), запускающий компилятор **DCC32.EXE**

```
rem файл MAKE_APP.BAT
rem КОМПИЛИРУЕМ ПРОЕКТ %1
if '%1' == '' goto exit
rem КОМПИЛИРУЕМ...
f:\Delphi\bin\DCC32.exe -B -L -Tf:\Delphi\bin %1.dpr
:exit
```

Пользователь может освежить знание отдельных опций командной строки компилятора, запустив его с пустой командной строкой.

Ниже приведен текст **BAT**-файла, последовательно обрабатывающий пять выполняемых файлов **MAGIC**, **MAGIC_E**, **MAGIC_A**, **MAGIC_B** и **MAGIC_S** путем последовательного обращения к ранее приведенному

```
rem файл MAKE_MAG.BAT
rem КОМПИЛИРУЕМ ФАЙЛЫ ПРОЕКТА 'MAGIC TOURS'
call make_app.bat magic
```

```
call make_app.bat magic_e
call make_app.bat maglc_a
call make_app.bat magic_b
call make_app.bat magic_s
```

13. ОСНОВНЫЕ ОТЛИЧИЯ СИНТАКСИСА C++Builder'а ОТ Delphi

В данном разделе приведены (в основном чисто формальные) отличия синтаксиса основных для систем **Delphi** и **C++Builder** языков программирования. В пределах этого раздела под '**Pascal**-ем' понимается **Object Pascal 8.0** для **Delphi**, термин '**C++**' указывает на версию основного для **C++Builder**'а языка программирования.

Для обращения к любой компонентной функции, свойству или переменной в **Pascal**'е используется оператор 'точка', например

```
Label1.Caption:='Hello, BAKANOV !';
```

В **C++** для указания на размещенные в 'куче' (*heap*) объекты служит оператор 'стрелка'

```
Label1->Caption="Hello, BAKANOV !";
```

а сами размещенные в 'куче' объекты создаются функцией **new** и принудительно уничтожаются функцией **delete**

```
TLabel *Label1 = new TLabel(0); // создать объект без
                                // указания владельца
delete Label1; // уничтожить объект, созданный без
               // указания владельца
```

В случае динамического создания объекта конструктору следует сообщить (в качестве параметра передается указатель **this**) информацию о владельце данного объекта (например, формы); в этом случае ответственность за разрушение объекта (вызов **delete**) берет на себя его владелец

```
TLabel *Label1 = new TLabel(this); // создать объект с указанием
                                   // владельца в виде  this
// delete Label1; // нет необходимости принудительно
                  // уничтожать объект, созданный с
                  // указанием владельца при создании
```

Для **Pascal**'я оператор присваивания суть сочетания двоеточия и знака ра-

венства, для **C++** - одинарный знак равенства.

Для **Pascal**'я оператор логического равенства суть одинарный знак равенства, для **C++** - двойной знак равенства.

В **Pascal**'е символьная строка заключается в одинарные кавычки, в **C++** - в двойные кавычки (а единичный символ - в одинарные кавычки).

В **Pascal**'е строчные и заглавные буквы эквивалентны, в **C++** - различаются. Обычно в **C++** имена переменных составляются из строчных (за исключением лидирующей) букв, имена констант начинаются со строчных букв. Например, присваивание **Labell->top=200;** вызовет ошибку **"Undefined symbol 'top'"** (следует писать **Labell->Top=200;**).

Pascal'евское ключевое слово **with** не имеет эквивалента в **C++**; вместо **as** в **C++** следует использовать оператор динамического преобразования типа **dynamic_cast**.

Вместо **Pascal**'вских множеств (**set**) в **C++** следует использовать битовые поля. В **Pascal**'е определен оператор конкатенации (слияния) строк (символ '+'), в **C++** программисту придется использовать функцию **strcat** (и подобные) для строк в стиле **C** и тот же '+' для **Pascal**-строк.

При работе в **C++** часто приходится использовать компонентную функцию **c_str()**, возвращающую **C**-подобный указатель (тип **char***) на **Pascal**-строку.

В **C++** каждая функция описывается с применением заключающих формальные параметры скобок (даже если список параметров пуст); таким образом, в **C++** нет присущего **Pascal**'ю разделения на функции и процедуры.

Обращение к элементу массива в **Pascal**'е описывается как [I,J], в **C++** как [I][J].

Эквивалентом **Pasca1**'евской конструкции **try/except** служит **try/catch** в **C++** (аналог **raise** суть **throw**).

Принадлежность некоей функции **ButtonlClick** классу **TForml** записывается в **Pascal**'е как **TForml.ButtonlClick(...)**, в **C++** как **TForml::ButtonlClick(...)**.

Заключение

Данное методическое руководство является фактически введением в предмет создания приложений с помощью **Delphi** / **C++Builder** и тем более WINDOWS-программирования вообще и позволяет пользователю освоить самые простые приемы разработки пользовательских программ для WINDOWS. Для повышения квалификации (чему практически нет ограничений) необходимо изучение литературных источников (часть из них приведена ниже) и, главное, постоянная практическая работа на ПЭВМ.

Список рекомендуемой литературы

1. Адлер М. Система WINDOWS: введение в программирование. Журнал 'МирПК', № 5 и след., 1991. с.36÷48.
2. Рубенкинг Н. Турбо-Паскаль для WINDOWS. -М., Мир, 1993. 536+552 с. (2 тома).
3. Фролов А.В.,Фролов Г.В. Мультимедиа для WINDOWS. Руководство программиста. -М., Диалог-МИФИ, 1994. -284 с. (Библиотека системного программиста, т. 15).
4. Фролов А.В.,Фролов Г.В. Программирование для WINDOWS'NT. Руководство программиста. -М., Диалог-МИФИ, 1997. -272/271 с. (Библиотека системного программиста, т. 26/27).
5. Федоров А.Г. Создание WINDOWS-приложений в среде Delphi. -М., КомпьютерПресс, 1995. -287 с.
6. Дантеманн Д.,Мишел Д.,Тейлор Д. Программирование в среде Delphi (пер. с англ.), DiaSoft Ltd., Киев, 1995. -608 с.
7. Матчо Дж.,Фолкнер Д.Р. Delphi (справочный материал). -М., БИНОМ, 1995. -464с.
8. Дарахвелидзе П.Г.,Марков Е.П. Delphi - среда визуального программирования. -СПтб., BHV-Санкт-Петербург, 1997. -352 с.
9. 0рлик С. Секреты Delphi на примерах. -М., БИНОМ, 1996. -316 с.
10. Федоров А.Г. Delphi 2.0 для всех. -М., КомпьютерПресс, 1997. -464 с.
11. Шамис В.А. Borland C++ Builder. Программирование на C++ без проблем. -М., Нолидж/Knowledge, 1997. -266 с.
12. Конопка Р. Создание оригинальных компонент в среде Delphi. -Киев, DiaSoft, 1996. -512 с.
13. Архангельский А.Я. Функции C++, C++Builder 5 и API Windows (справочное пособие). -М., Бином, 2000. -240 с.
14. Калверт Ч., Калверт М., Кастер Дж. Borland Kylix (руководство разработчика). Диалектика, 2002.